# Docker Handbuch für Einsteiger

*Der leichte Weg zum Docker-Experten*

# Inhaltsverzeichnis

## 16. Docker Stack 309



## 17. Kubernetes 325

Kapitel 1

# Einleitung



## 1.1 Vorwort

Herzlich willkommen in der Welt des Cloud-Computing mit Docker, Docker Swarm und Kubernetes. Es ist eine abenteuerliche Welt, die sich so rasant fortentwickelt, wie das bisher in der IT-Welt noch nie der Fall war. Experten, die sich in dieser Welt auskennen, sind daher in allen Branchen gefragt. Wenn sie also Ihre berufliche Zukunft verbessern wollen, dann sollten Sie sich im Umgang mit Docker und Kubernetes oder ähnlichen Technologien vertraut machen.

Es fließen im Umfeld von Docker so viele IT-Themen und Sachgebiete zusammen, dass man sich schon überwältigt fühlen könnte. Andererseits wird jeder in seinem Aufgabenbereich nur einen Teil der Möglichkeiten in der Praxis nutzen. Frontend-Entwickler werden zum Beispiel andere Schwerpunkte haben als Backend-Entwickler, DevOps-Entwickler oder Programmierer im Test-Umfeld.

Dieses Buch soll aber kein umfangreiches Nachschlagewerk sein, sondern es wird ein Überblick über Zusammenhänge von verschiedenen Themen angeboten, die in Verbindung mit Docker und Kubernetes von Bedeutung sind. Dabei wird bei diesen Themen aber nicht in die Tiefe gegangen. Es ist auch so, dass die Kapitel in diesem Buch aufeinander aufbauen. Es werden zu Beginn einfache und leicht verständliche Beispiele vorgestellt, die in nachfolgenden Kapiteln Schritt für Schritt erweitert werden.

Wie gesagt: Die Entwicklung im Umfeld des Cloud-Computing schreitet in einer solchen Geschwindigkeit voran, dass es nicht einfach ist, da Schritt zu halten.

*Ständige Veränderungen sind heute die neue Normalität! Damit muss man sich in Zukunft als Entwickler abfinden.*

***Das hat auch Konsequenzen für dieses Buch. Die eine oder andere Information in diesem Buch ist wahrscheinlich schon dann wieder überholt, wenn es frisch aufgelegt und im Handel ist. Das gilt vor allem für Screenshots von Internetseiten und auch von Dialog- oder Programmfenstern.***

Bis zuletzt habe ich bei der Arbeit an diesem Buch Screenshots ersetzt und Beschreibungen angepasst. Die Änderungen sind im Normalfall nicht so gravierend, dass man den Beschreibungen nicht mehr folgen kann. Meist ist nur der Aufbau von Fenstern oder Webseiten anders, es gibt neue Steuerelemente oder Steuerelemente verschwinden. Gelegentlich ändert sich auch die Reihenfolge von Aktionen, die dort durchgeführt werden sollen.

## 1.2 Die Microservice Revolution

Microservice-Architekturen in Kombination mit der Virtualisierung von Systemen durch Container sind in den letzten Jahren wie eine Flutwelle über die Softwareentwicklung und auch über die Entwicklerteams hereingebrochen. Es haben sich ganz neue Technologien und Tools etabliert, ganz neue Möglichkeiten sind sichtbar geworden. Die Sub-Systeme werden unabhängiger voneinander und sogar unabhängiger von den Betriebssystemen und von der Hardware, auf denen sie ausgeführt werden.

Den entscheidenden letzten Schub erhielt die Idee der Microservices durch die Veröffentlichung von Docker, was zur Folge hatte, dass die vereinfachte Nutzung auf Container basierten, virtuellen Systemen einer noch größeren Gemeinschaft von Entwicklern zugängig gemacht wurde.

Die Grundidee der Microservice Architektur ist nicht neu. Seit vielen Jahren hat sich in der Softwareentwicklung die Erkenntnis durchge-

setzt, dass es besser ist, ein System in viele kleine Komponenten aufzuteilen, von denen jede genau eine Aufgabe perfekt erledigt (do one thing and do it well), als ein großes monolithisches System zu erstellen, das alles kann (die berühmte eierlegende Wollmilchsau).

Damit wird die Wiederverwendbarkeit von Softwarekomponenten drastisch erhöht. Gleichzeitig ist es leichter, solche Systeme zu debuggen, zu warten und zu erweitern.

Auch Microservices folgen diesem Architekturmuster, bei dem komplexe Softwaresysteme aus Komponenten zusammengesetzt werden, die voneinander unabhängig, also entkoppelt, sind.

Neu bei den Microservices ist die hohe Flexibilität. Für die Ausführung jedes Services wird jeweils ein eigener Prozess zur Verfügung gestellt. Die Kommunikation zwischen diesen Prozessen erfolgt über sehr schlanke Schnittstellen, die auch noch unabhängig von der verwendeten Programmiersprache sind.

Entwickler werden bei der Entwicklung von Microservices verstärkt dazu bewegt, die Systeme in kleinere Komponenten aufzubrechen und diese voneinander zu entkoppeln.

Wird ein System beim Entwurf in viele Microservices aufgebrochen, dann ist es möglich, dass diese dezentral und unabhängig voneinander von verschiedenen Teams entwickelt und verteilt werden. Auch die Skalierung der beteiligten Services ist unabhängig vom Gesamtsystem.

Durch die Einführung der Container-Technologie mit virtuellen Maschinen und virtuellen Servern hat man die Voraussetzungen geschaffen, dass Microservices überall im Web und unabhängig von den verfügbaren Plattformen ausgeführt werden können. Es ist so auch möglich, die Systeme gegen Ausfall anderer Services abzusichern.

Noch ein Vorteil ist, dass „Continous Delivery“ durch die Aufteilung in kleinere Services einfacher wird.

Als Nachteil muss hier allerdings eine erhöhte Komplexität beim Testen der Software, beim Verteilen der Software, beim Logging und beim Monitoring in Kauf genommen werden. Die Fehlertoleranz dieser Systeme sinkt meist und die Last im Netzwerk steigt.

Insgesamt kann man aber sagen, dass mit der Einführung von Microservice-Architekturen die damit entwickelten Applikationen schneller entwickelt werden können. Sie sind leichter zu warten, robuster und von insgesamt höherer Qualität.

## 1.3 Das Ziel dieses Buches

Dieses Buch wurde für verschiedene Lesergruppen konzipiert.

Da sind in erster Linie die Entwickler, die mit Docker flexible Webservices oder Applikationen entwickeln möchten. Dabei soll auf der einen Seite dem Docker-Neuling ein verständlicher und leicht nachvollziehbarer Einstieg in die Dockerwelt geboten werden. Es werden zunächst die Begriffe aus dem Docker-Umfeld erklärt. Es folgt eine Schritt-für-Schritt-Anleitung zur Installation von Docker unter Windows. Die Installation unter MacOS und Linux wird im Anhang beschrieben. Dann gibt es eine Übersicht über die Funktionen und Einsatzmöglichkeiten von Docker. Die praktische Handhabung wird mit einfachen Beispielen demonstriert. Zusätzliche Aufgaben im Anschluss an ein Thema helfen dabei, das neu Erlernte zu vertiefen.

Wir geben Ihnen eine Übersicht über die vorhandenen Images im Docker Hub. Dabei stellen wir einige nützliche Images, die häufig zum Einsatz kommen, genauer vor.

Ein eigenes Kapitel ist der Übersicht über verfügbare nützliche Tools für die Arbeit mit Docker gewidmet. Die Funktionalität dieser Werkzeuge wird kurz beschrieben und es wird auch gezeigt, wie sie die Arbeit mit Docker unterstützen bzw. vereinfachen.

Um die Funktionsweise von Docker besser zu verstehen, werden im Kapitel über die Docker-Architektur die verschiedenen Docker-Komponenten genauer beschrieben. Das Verständnis für Struktur und die Abhängigkeiten der Komponenten wird vertieft und durch Grafiken veranschaulicht.

Falls Sie als Entwickler schon Erfahrung mit Docker gesammelt haben, wird der Teil des Buches mit den fortgeschrittenen Techniken der interessantere für Sie sein. Dort werden Ihre Docker-Kenntnisse vertieft und weitere Tools vorgestellt. Dazu gehören Themen wie die Orchestrierung von Docker Containern mit Docker Compose, der Einsatz von Docker Swarm und Docker Stack.

Als umfangreicheres praktisches Beispiel entwickeln wir mit Ihnen schrittweise eine WordPress-Blog-Anwendung. Dabei werden die benötigen Funktionalitäten als Microservices entworfen, auf mehrere Container aufgeteilt und mit Docker Compose verwaltet. Später stelle ich vor, wie man solche Anwendungen mit Docker Swarm verteilt.

Zahlreiche Tipps und Tricks aus der Praxisarbeit mit Docker dürfen in diesem Buch natürlich auch nicht fehlen.

Schließlich stelle ich Ihnen in diesem Buch noch vor, wie bei Docker-Anwendungen die Bereitstellung, Skalierung und Verwaltung mithilfe von Kubernetes automatisiert werden kann.

Neben Informationen für Web-Entwickler enthält dieses Buch auch Informationen für Softwarearchitekten, Projektleiter und andere Entscheidungsträger. Diese sind bei Entscheidungen zur Auswahl aus möglichen Technologien und verfügbaren Tools recht nützlich.

Nicht zuletzt möchte ich hier noch die Studierenden verschiedener Fachrichtungen erwähnen, die mit dem Einsatz von Containertechnologie neue und innovative Anwendungen im Web entwickeln möchten.

## 1.4 Konventionen im Buch

Texte, die in der Kommandozeile einer Shell (z.B. PowerShell oder einem Linux TTY) eingegeben werden, sind in `Schreibmaschinenschrift` (Courier) gesetzt. Das gleiche gilt für Texte die als Programm-Quellcode oder als Skript-Anweisungen eingegeben werden. Auch der Inhalt von Dockerfiles und Docker-Compose-Dateien ist in Courier formatiert.

Shell-Kommandos für die Windows PowerShell werden mit dem führenden Zeichen > angegeben. Bei anderen Shells wird $ oder # so verwendet, dass es jeweils zu den Screenshots der vorgegebenen Beispiele passt.

```
> docker image ls
$ sudo apt-get update
# cat hello.txt
```

Dieses Zeichen wird nicht mit eingetippt. Es steht als Platzhalter für das System Prompt.

Um umfangreichere Kommandos übersichtlicher zu gestalten, werden diese im Buch mehrzeilig, als mit Zeilenumbruch, dargestellt. Bei der mehrzeiligen Darstellung von Shell Kommandos wurden hier Backticks ( ` ) zum Maskieren des Zeilenendes verwendet.

Beispiel:

```
> docker run -i -t '
--name=voltest '
--mount source=test-vol,target=/test_data '
ubuntu /bin/bash
```

Dies ist die Variante für die Eingabe in einer PowerShell. Bei anderen Shell-Applikationen, wie zum Beispiel der Ubuntu Shell, ist das Zeichen zum Maskieren von Sonderzeichen wie einem Zeilenende in der Regel der Backslash ( \ ).

Falls Sie also nicht mit der PowerShell arbeiten, dann ersetzen sie das Backtick-Zeichen aus den entsprechenden Beispielen durch einen Backslash.



Beispiel:

```
> docker run -i -t \
--name=voltest \
--mount source=test-vol,target=/test_data \
ubuntu /bin/bash
```

Dateinamen oder andere Namen, die im System vergeben werden können (z.B. Namen von Datenbanken oder Tabellen), sind ebenfalls in der Schriftart Courier gehalten und werden zwischen einfache Anführungszeichen gesetzt.

Beispiel:

```
'docker-compose.yaml'
```

Textverweise auf Elemente von Benutzeroberflächen wie Fenster oder Webseiten, wie zum Beispiel Menübefehle, Schaltflächen und Steuerelemente, sind als KAPITÄLCHEN dargestellt. Die Angabe

DATEI | NEU | PROJEKT

bedeutet, dass sie aus dem Hauptmenü den Menüpunkt DATEI auswählen, aus dem Drop-Down-Menü den Befehl NEU und dort aus dem Untermenü die Auswahl PROJEKT.

Die Angabe [HILFE] weist auf eine Schaltfläche mit dem Label [Hilfe] hin.

Optionen in Dialogfeldern sind *Kursiv* formatiert.

Namen von Fenstern, wie zum Beispiel Dialogfenster, werden als Kapitälchen zwischen „ANFÜHRUNGSZEICHEN“ gesetzt.

Eine vollständige Beschreibung der Kommandos für die Docker CLI sowie die ausführliche Beschreibung der Dockerfile-Anweisungen finden Sie im Anhang dieses Buches.

## 1.5 Warum braucht man Docker?

Wenn man eine Anwendung entwickelt, dann möchte man diese so vielen Anwendern wie möglich zugänglich machen. Aber diese Anwender nutzen verschiedene Ausführungsumgebungen, das heißt verschiedene Hardware mit unterschiedlichen Betriebssystemen, die dann auch noch verschiedene Versionen haben können.

Dann sollen in der Regel auf diesen Systemen auch noch mehrere Anwendungen lauffähig sein. Die nutzen dann meistens Laufzeitbibliotheken, Tools oder Datenbanken, die es auch wieder in mehreren Versionen gibt und die natürlich untereinander, mit den installierten Anwendungen und mit der aktuellen Betriebssystemversion kompatibel sein müssen.

Diese Situation hat in der Vergangenheit regelmäßig zu Problemen geführt, die nur mit viel Aufwand gelöst werden konnten. Administratoren haben in diesem Zusammenhang schon von der ‚Konfigurationshölle' oder dem ‚DLL Versions-Alptraum' gesprochen.

Um die Verteilung von Anwendungen zu vereinfachen, sind als Lösung für diese Probleme sogenannte Container-Plattformen entwickelt worden. Eine davon ist Docker, das im Moment wahrscheinlich beliebteste und am weitesten verbreitete Container-System.

## 1.6 Was muss ich mir unter Docker vorstellen?

Docker ist ein Tool, welches die Entwicklung, Verteilung und Ausführung von Anwendungen durch die Nutzung von Containertechnologie vereinfacht.

Eine Applikation kann dabei mit allen Bestandteilen, die sie braucht, zusammengepackt werden. Dazu gehören zum Beispiel Bibliotheken, Datenbanken, Treiber oder auch Konfigurationsdateien.

Docker ist also eigentlich nichts weiter als eine Ansammlung von Produkten. Diese kommen aus dem Umfeld von , Plattform-as-a-Service' (PaaS).

Durch den Einsatz virtueller Betriebssysteme, die auf den Ziel-Plattformen laufen, wird es dann möglich, Serveranwendungen als komplette Pakete, sogenannte ‚Container', auszuliefern.

Docker Container sind dadurch systemunabhängig. Ob Anwendungen unter Windows, Linux oder MacOS ausgeführt werden, spielt keine Rolle mehr, wenn diese in Docker Container verpackt sind und durch ein virtuelles System ausgeführt werden.

Server-Applikationen können in Docker Containern fertig installiert und mit diesen verteilt werden. Aufwendige und zeitraubende Setup- oder Install-Aktivitäten, wie wir sie bisher kennen, bleiben den Administratoren damit erspart.

Ein weiterer Vorteil dabei ist, dass vorhandene Server-Applikationen, die bisher direkt aus einem Betriebssystem heraus ausgeführt wurden, ohne Weiteres in Docker Container überführt werden können. Dort laufen diese, ohne dass der Quellcode geändert oder angepasst werden muss.

## 1.7 Was ist Docker nicht?

Nur um Missverständnissen vorzubeugen: Docker Container sind nicht für die Ausführung von typischen Client-Anwendungen gedacht (z.B. Word oder Power Point). Das können virtuelle Maschinen wie zum Beispiel ‚Vitual Box' oder VMware besser.

In Docker Containern werden also ausschließlich Server-Anwendungen ausgeführt, die über Schnittstellen mit Protokollen wie HTTP kommunizieren.

Man kann sich Docker ein wenig wie eine Virtuelle Maschine vorstellen (z.B. Virtual Box), aber Docker ist viel leichtgewichtiger als eine Virtuelle Maschine.

## 1.8 Entwicklungsgeschichte

Docker ist jetzt 7 Jahre alt und in der Zwischenzeit ist recht viel passiert. Es folgt an dieser Stelle eine Übersicht über die wichtigsten Meilensteine in der Entwicklungsgeschichte von Docker.

- Die Firma ‚Docker Inc.' wurde im Sommer 2010 von Solomon Hykes und Sebastian Pahl gegründet. Hykes startete damals das Docker-Projekt in Frankreich innerhalb von ‚dotCloud', einer ‚PaaS' (Platform as a Service) Firma.
- Docker wurde erstmals 2013 während der Python Conferece (PyCon) in Santa Clara vorgestellt.
- Im März 2013 wurde Docker als Open-Source-Software freigegeben (released).
- Zunächst lief Docker noch auf LXC, einem virtuellen Linux Container.
- Ein Jahr später hat man LXC durch eigene Docker-Komponenten ersetzt. Diese wurden in der Programmiersprache GO entwickelt. GO oder auch Golang ist eine Programmiersprache, die von Robert Grisemeier bei Google entworfen wurde. GO ähnelt sehr der Programmiersprache C, bietet aber Speicher-Sicherheit, Garbage Collection und Typ-Sicherheit.
- Im September 2013 gaben Docker und Red Hat ihre Zusammenarbeit im Umfeld von ‚Fedora Linux', ‚Red Hat Enterprice Linux' und ‚Open Shift Container' bekannt.

- Danach, im Oktober 2013, kündigt dotCloud an, dass es sich in Docker umbenennt.
- Amazon gibt im November 2014 den Einsatz von Docker Container Services im Umfeld der ‚Amazon Elastic Compute Cloude' (EC2) bekannt. EC2 ist der zentrale Bestandteil der Amazon Web Services (AWS).
- Microsoft integriert im Oktober 2014 die Docker Engine in ‚Windows Server'.
- Docker gibt im November 2014 seine Partnerschaft mit Stratoscale bekannt.
- Im Dezember 2014 folgt Partnerschaft von Docker mit IBM. Dadurch wird die Voraussetzung für eine bessere Integration der IBM Cloud mit Docker geschaffen.
- Docker, in Zusammenarbeit mit verschiedenen anderen Organisationen, erklären im Juni 2015, dass sie an einem neuen Standard für Software Container arbeiten, der unabhängig von Herstellern und Betriebssystemen sein soll.
- Im Herbst 2016 ist Docker zum ersten Mal im ‚Native Mode' (d.h. ohne zusätzliche externe Software Layer) unter ausgewählten Windows Versionen verfügbar.
- Microsoft gibt im Mai 2019 die Version 2 von WSL, dem Windows Subsystem für Linux, heraus. Es handelt sich dabei um eine Kompatibilitätsschicht zur Ausführung von LINUX-Applikationen unter Windows 10. Docker Inc. beginnt danach mit der Entwicklung einer Docker-Version für Windows, die auf der Basis von WSL 2 läuft.
- Im Oktober 2019 wird bekannt, dass Docker finanzielle Probleme hat. Trotz zahlreicher Erfolge hat Docker es wohl nicht geschafft, wirtschaftlich erfolgreich zu sein.
- Die bisher neueste Meldung kommt Mitte November 2019 – Docker verkauft die Docker Enterprise Sparte an den Cloud-Dienstleister Mirantis. Docker Inc. erklärt dazu, dass man sich wieder mehr auf Docker Hub und Docker Desktop konzentrieren will.

Mit dem Verkauf wurde auch der CEO von Docker ausgetauscht. Den Posten übernimmt der bisherige CPO Scott Johnston von Rob Bearden.

Kapitel 2

# Docker-Begriffe

## 2.1 Was ist ein Container?

Ein Container vereint in sich Software, zusammen mit zugehörigen Bibliotheken, Tools und Konfigurationsdateien. Applikationen laufen so schnell und zuverlässig auf verschiedenen Umgebungen.

Verschiedene Container sind entkoppelt und voneinander unabhängig, aber sie können über wohldefinierte Kommunikationskanäle untereinander Informationen austauschen.

Durch die Entkopplung von Containern können die Unverträglichkeiten von Bibliotheken, Tools oder Datenbanken umgangen werden, wenn diese von den Applikationen in unterschiedlichen Versionen benötigt werden.

## 2.2 Was ist ein Container Image?

Container Images werden genutzt, um zur Laufzeit Container-Instanzen zu erzeugen. Bei Docker werden Docker Images zu Docker Containern, wenn sie auf einer Docker Engine als Prozess ausgeführt werden.

Man kann sich ein Container Image wie eine Kopiervorlage vorstellen, die genutzt wird, um davon Container als Kopien herzustellen.

Diejenigen, welche in Objektorientierter Programmierung bewandert sind, können sich das Image als Klasse vorstellen und die Container als Objekte dieser Klasse.

Server Applikationen werden innerhalb des Container Prozesses ausgeführt. Egal ob es sich dabei um Windows- oder Linux-Applikationen

handelt: Sie werden immer gleich ausgeführt, unabhängig von der jeweiligen Infrastruktur.

## 2.3 Das Dockerfile



Das Dockerfile ist eine Textdatei, welche im Grunde genommen Linux-Kommandos enthält, die ein Anwender auch auf der Linux-Kommandozeile eingeben könnte. Im Dockerfile erledigen diese Kommandos alle Aufgaben, die nötig sind, um ein Docker Image zusammenzustellen.

Wird das Kommando `docker build` ausgeführt, führt das zur automatischen Ausführung eines Dockerfiles, aus dem die Kommandozeilen Anweisungen hergenommen werden, um damit ein Image zu erstellen.

Das allgemeine Format von Anweisungen in einem Dockerfile sieht folgendermaßen aus:

```
1 # Comment
2 INSTRUCTION <arguments>
```

### 2.3.1 Dockerfile-Elemente

Es folgt eine kurze Übersicht mit den wichtigsten Elementen und Anweisungen, die in Dockerfiles verwendet werden können.

*Kommentare:*

Kommentare werden im Dockerfile mit dem Doppelkreuz-Zeichen # eingeleitet.

*FROM-Anweisung:*

Die erste Anweisung im Dockerfile muss die Anweisung `FROM <imagename>[:<tag>]` sein. Diese Anweisung bestimmt, welches Parent Image verwendet werden soll, um davon das neue Image abzuleiten. Als Parameter wird hier ein Imagename angegeben. Optional kann noch ein Tag hinzugefügt werden, durch das die Version eines

Images bestimmt werden kann. Lässt man das weg, wird automatisch die neueste Version als Vorlage verwendet.

Beispiel:

```
1 FROM ubuntu
```

*RUN-Anweisung:*

Die RUN-Anweisung führt beliebige Kommandos auf einer Ebene oberhalb des aktuellen Images aus und stellt das Ergebnis zur Verfügung.
Es stehen zwei Varianten des RUN-Kommandos zur Verfügung. Die eine Variante ist die ‚Shell Form', bei der das Kommando in einer eigenen Shell ausgeführt wird (`sh` unter Linux oder `cmd` unter Windows):
`RUN <kommando>`
Die zweite Variante ist die ‚exec Form'. Dabei werden als Argument der Name des Executables und die Parameter in eckigen Klammern übergeben:
RUN [`"executable", "param1", "param2"`]

Beispiel:

```
1 RUN chmod +x ./hello.sh
2 RUN [" python", "hello.py"]
```

*CMD-Anweisung:*

Die CMD-Anweisung ist der RUN-Anweisung sehr ähnlich. Aber im Gegensatz zur RUN-Anweisung kann die CMD-Anweisung nur einmal, als letzte Anweisung des Dockerfiles, ausgeführt werden. Stehen mehrere CMD-Anweisungen im Dockerfile, wird nur die letzte ausgeführt, die anderen werden ignoriert.
Bei der CMD-Anweisung gibt es drei Varianten. Auch hier gibt es die ‚Shell Form' und die ‚exec Form'. Die dritte Variante ist eine parametrisierte Form, bei der die default Parameter des Entrypoints verwendet werden.
`RUN ["param1", "param2"]`

*COPY-Anweisung:*

Die COPY-Anweisung kopiert Dateien, Verzeichnisse vom Host Rechner, die mit Parameter `<src>` angegeben werden, und fügt sie auf dem Dateisystem des Images dem Verzeichnis zu, das mit `<dest>` angegeben wird.
COPY `<src> <dest>`.



Beispiel:

```
1 COPY html /usr/share/nginx/html
```

## 2.4 Was ist die Docker Engine?

Die Docker Engine stellt die Laufzeit-Umgebung für Container zur Verfügung und läuft auf Linux, macOS und Windows Server Betriebssystemen.

Der Einsatz von Docker Engine ermöglicht, dass Container-Anwendungen auf jeder Infrastruktur problemlos laufen können.

## 2.5 Wer ist der Container Host?

Als Container Host (Gastgeber) bezeichnet man den Computer, der die Container Engine ausführt.

## 2.6 Was sind Container-Netzwerke?

Der Container Host stellt seinen Docker Containern Netzwerke zur Verfügung, über die Container miteinander oder mit Client-Anwendungen kommunizieren können.

Es gibt bei Docker vier verschiedene Netzwerk Arten:

- Closed Network / none Network
- Bridge Network

- Host Network
- Overlay Network

Eine genauere Beschreibung dieser Netzwerk Arten folgt im Kapitel ‚Netzwerke und Docker' (Kapitel 10).

## 2.7 Was ist die Container Registry?

Die Container Registry ist eine Server seitige Applikation, die es erlaubt, Docker Images zu speichern und bereitzustellen.

Eine Registry ist eine Stelle zum Auffinden von Images. Docker Hub ist so eine Registry. Es gibt aber auch Registries außerhalb von Docker Hub, zum Beispiel stellt Google die „Google Container Registry" zur Verfügung.

Eine Container Registry sollte verwendet werden, wenn der Speicherort und die Verteilung von Images genau überwacht werden sollen.

Es gibt die Möglichkeit sowohl öffentliche als auch private Registries zu nutzen.

## 2.8 Was ist der Docker Hub?

Der Docker Hub ist ein auf Cloud-Technologie basierter Repository Service, den Docker-Anwender und -Partner nutzen können, um Container Images abzulegen und zu verwalten.

Alle Docker Tools greifen Standardmäßig auf den Docker Hub zu, der deswegen das wichtigste öffentliche Docker Repository darstellt.

Es werden bei Docker Hub sowohl öffentliche als auch private Repositories abgelegt und zur Verfügung gestellt.

Der Docker Hub bietet aber noch mehr als nur Registry Funktionalität. Er erlaubt auch die Images von Organisationen zu verwalten und den Zugriff durch Mitglieder dieser Organisationen zu regeln.

## 2.9 Was ist der Unterschied zwischen Containern und Virtuellen Maschinen?

Container und Virtuelle Maschinen ähneln sich, was die Isolation und Zuweisung von Ressourcen angeht.



Sie unterscheiden sich aber dadurch, dass Container lediglich Betriebssysteme virtualisieren, keine Hardware. Ein Container läuft auf der echten Hardware eines Hosts und nicht auf einer virtuellen, simulierten Hardware.

Alle Container werden innerhalb eines einzelnen Betriebssystem Kernels ausgeführt. Das macht sie leichtgewichtiger als Virtuelle Maschinen.

Virtuelle Maschinen emulieren ein komplettes Computersystem mit seiner Architektur und bieten die Funktionalität eines physischen Computersystems mit seiner Hard- und Software.

Kapitel 3

# Vorbereitung



## 3.1.1 Docker Desktop Installation

Bei ‚Docker Desktop' handelt es sich um Applikationen für Windows und MacOS, mit deren Hilfe recht einfach und komfortabel fertige Container-Anwendungen erstellt werden können. Dabei können beliebige Frameworks, Programmiersprachen und Zielplattformen zum Einsatz kommen.

## 3.1.2 Docker Desktop für Windows Installieren

### *3.1.2.1 Systemvoraussetzungen*

Voraussetzung für eine erfolgreiche Docker-Installation ist ein Computer, auf dem Windows 10 Professional oder Windows 10 Enterprise als 64 Bit Version installiert ist.

### *3.1.2.2 Download des Installationsprogramms*

Das Docker-Installationsprogramm kann über die Docker-Homepage heruntergeladen werden.

Hier der Link auf diese Seite:

https://www.docker.com/

Der folgende Screenshot zeigt die Startseite von Docker. Diese Seite kann zu einem anderen Zeitpunkt natürlich etwas anders aussehen.

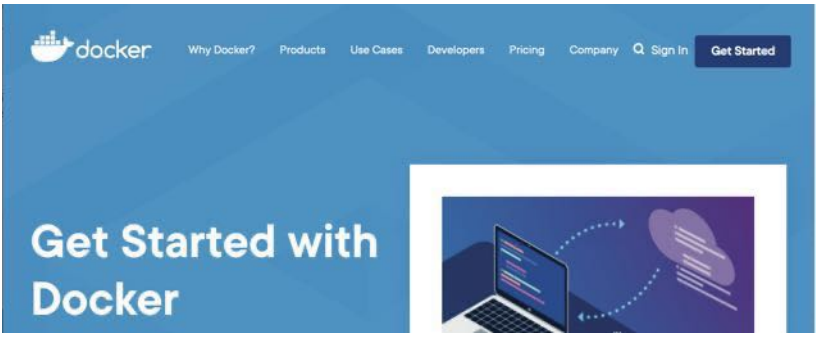



**Abb. 3.1** Die Startseite von Docker im Internet

Bewegen Sie auf dieser Seite den Mauszeiger über den Menütext [PRODUCTS]. Dadurch öffnet sich das folgende Untermenü.

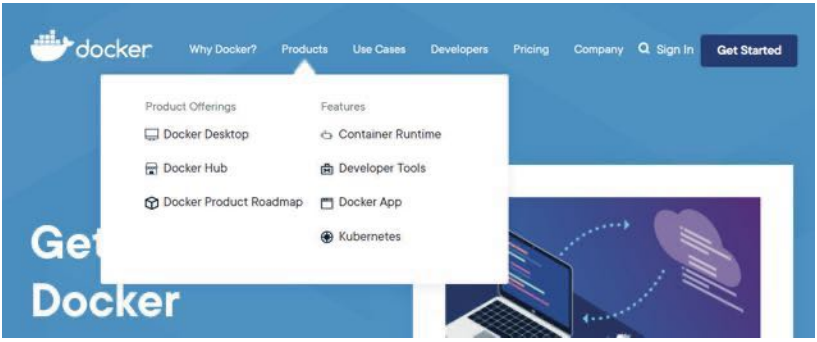



**Abb. 3.2** Docker-Produkte auf der Homepage

In diesem Untermenü den Menüpunkt [DOCKER DESKTOP] mit der Maus anklicken. Es wird zur Download Seite von Docker Desktop weitergeleitet.


**Abb. 3.3** Download-Seite von Docker Desktop für Mac und Windows

Klicken Sie jetzt den Button [Download for Windows].

Sie werden danach auf eine neue Webseite weitergeleitet, über die man sich bei Docker registrieren kann.

Die Registrierung starten Sie auf dieser Seite über den Button [Please Sign in to Download]. Folgen Sie auf den nächsten Seiten den Anweisungen, um Ihre Registrierung durchzuführen. Sie geben dabei Ihre Docker ID an und ein Passwort. Bitte merken Sie sich diese Angaben gut. Wir werden sie im Laufe der weiteren Praxisübungen immer wieder benötigen.

Falls Sie schon registriert sind, erscheint die folgende Anmeldeseite. Geben sie hier Ihre Docker ID und das Passwort ein und aktivieren Sie die Schaltfläche [Sign In], um sich anzumelden.

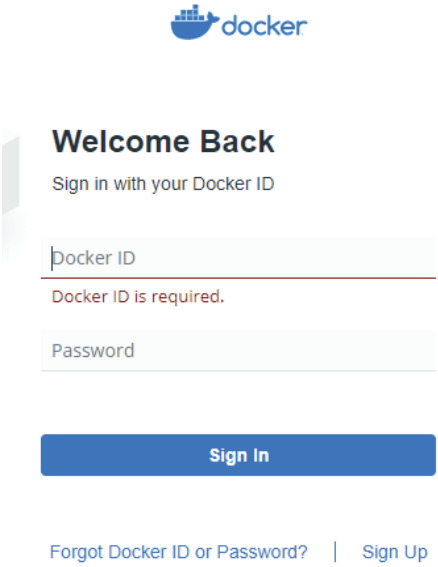



**Abb. 3.4** Docker-Anmeldeseite

Nach erfolgreicher Anmeldung landet man auf der Webseite ‚Download and Take a Tutorial'.


**Abb. 3.5** Docker-Seite ‚Download and Take a Tutorial'

Wir aktivieren den Button [Get started with Docker Desktop], um auf die ‚Quick Start' Seite zu kommen.


**Abb. 3.6** Docker-Webseite ‚Quickstart'

Jetzt muss nur noch der Button [Download Docker Desktop for Windows] aktiviert werden und der Download wird gestartet.

### *3.1.2.3 Installation von Docker*

Nach vollständigem Download kann die heruntergeladene Datei ‚Docker Desktop Installer.exe' ausgeführt werden, um die Installation zu beginnen.

Als Erstes erscheint der Installationsdialog.

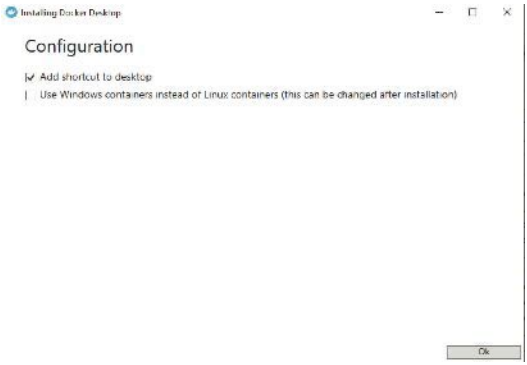



**Abb. 3.7** Installations-Dialog für Docker Desktop

Sie können hier per Check Box bestimmen, ob ein Desktop Icon zum Start von ‚Docker Desktop' automatisch erstellt werden soll.

Eine zweite Option ermöglicht die Auswahl einer Voreinstellung, ob Windows Container oder Linux Container genutzt werden sollen.

Der Unterschied und die Vor- und Nachteile der beiden Containerarten werden später in diesem Buch erläutert.

Behalten Sie im Moment die Voreinstellung bei. Diese Option kann zu einem späteren Zeitpunkt jederzeit verändert werden.

Nach Klicken des [OK] Buttons wird ‚Docker Desktop' installiert. Die Installation kann eine ganze Weile dauern.

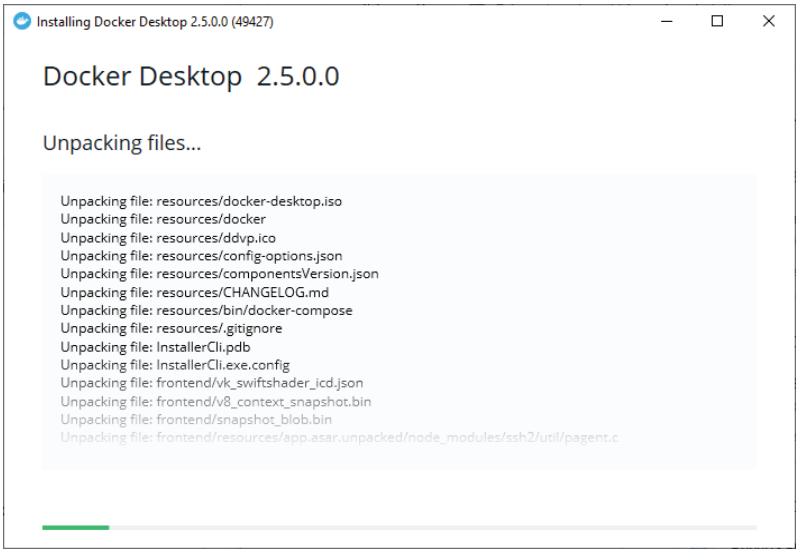



**Abb. 3.8** Docker Desktop wird installiert

Nach der Installation muss der Rechner neu gestartet werden.

Falls Hyper-V und Container noch nicht gestartet sind, erscheint die folgende Meldung:

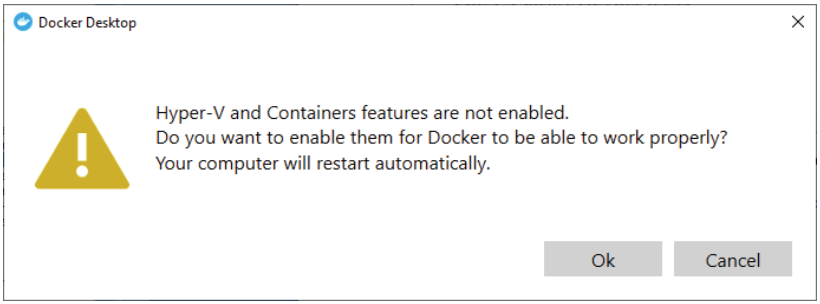



**Abb. 3.9** Docker-Meldung Hyper-V and Container are not enabled

Klicken sie hier auf den [OK] Button, um sie zu aktivieren.

Der Computer wird danach noch einmal neu gestartet.

Um Docker zu nutzen, muss der aktuell angemeldete Benutzer Mitglied der Benutzergruppe DOCKERUSERS sein. Die Gruppe wird bei der Installation angelegt und der bei der Installation angemeldete Benutzer wird automatisch dieser Gruppe als Mitglied zugefügt.

Bevor ein anderer Benutzer Docker verwenden kann, muss dieser der Gruppe DOCKER-USERS zugefügt werden. Andernfalls erscheint die Fehlermeldung

„Access Denied, you are not allowed to use Docker...“

Wir überprüfen jetzt, ob Docker Desktop korrekt gestartet wurde. Öffnen Sie folgendermaßen die Windows PowerShell.

Geben sie dazu im Suchfeld des Startmenüs den Text „PowerShell“ ein (Abb. 3.10).

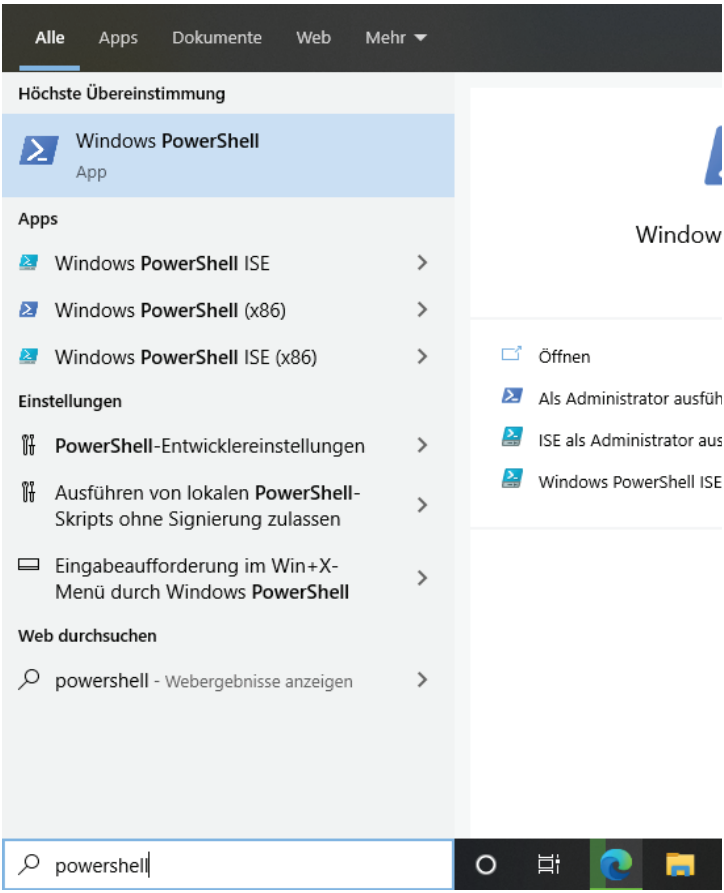



**Abb. 3.10** Start der Windows PowerShell

In der Liste mit den angezeigten Suchergebnissen klicken Sie auf Windows PowerShell, um die Shell normal zu starten.

Klicken Sie mit der rechten Maustaste darauf und wählen Sie Als Administrator ausführen aus, um alle Funktionen dieser Eingabekonsole zu verwenden.

Jetzt erscheint das Fenster der ‚PowerShell' Eingabe Konsole (Abb. 3.11):

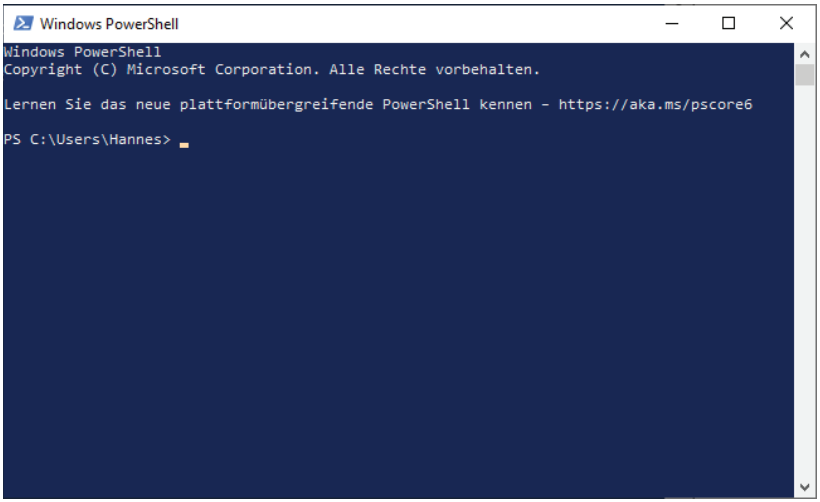



**Abb. 3.11** Das Fenster der Windows PowerShell

Als ersten Test geben wir hier das folgende Kommando ein:

```
> docker
```

Es wird eine Liste aller Docker-Kommandos angezeigt.

Um die aktuelle Version von Docker zu erhalten, geben wir nun ein:

```
> docker version
```

In der Konsole werden die Versionsinformationen der aktuell installierten Docker-Anwendung ausgegeben.

Herzlichen Glückwunsch, Sie haben ‚Docker Desktop' erfolgreich installiert.

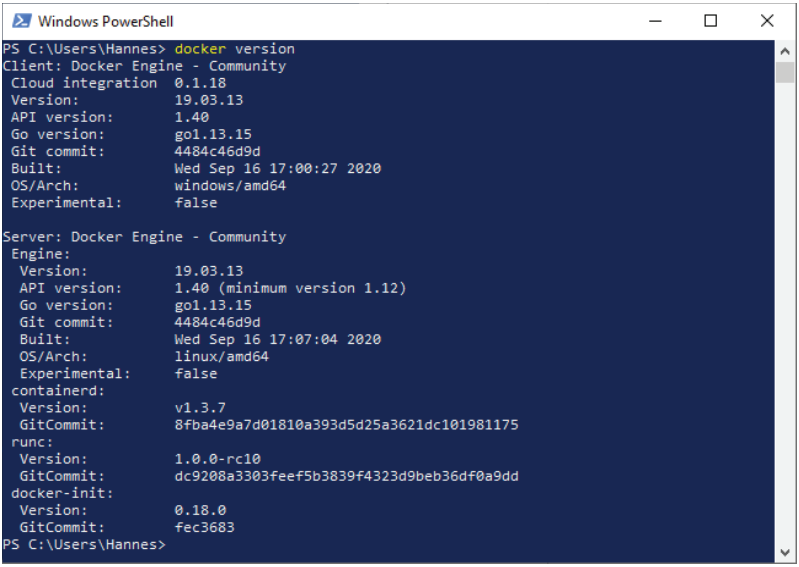



**Abb. 3.12** Das Kommando ‚docker version'.

### 3.1.3 Andere Betriebssysteme

Die Installation für MAC-OS und Linux wird im Anhang vorgestellt.

## 3.2 Erste Versuche mit Docker

### 3.2.1 Docker Desktop starten

Wenn Docker als Systemdienst installiert ist, wird Docker Desktop automatisch gestartet.

Ist dies nicht der Fall, dann suchen Sie den Menüpunkt Docker im Startmenü und klicken diesen mit der Maus an, um ‚Docker Desktop' zu starten. Alternativ kann man auch den Suchbegriff ‚Docker' im Suchfeld des Windows Startmenüs eingeben und dann in der Ergebnisliste den Eintrag Docker Desktop auswählen (Abb. 3.13).

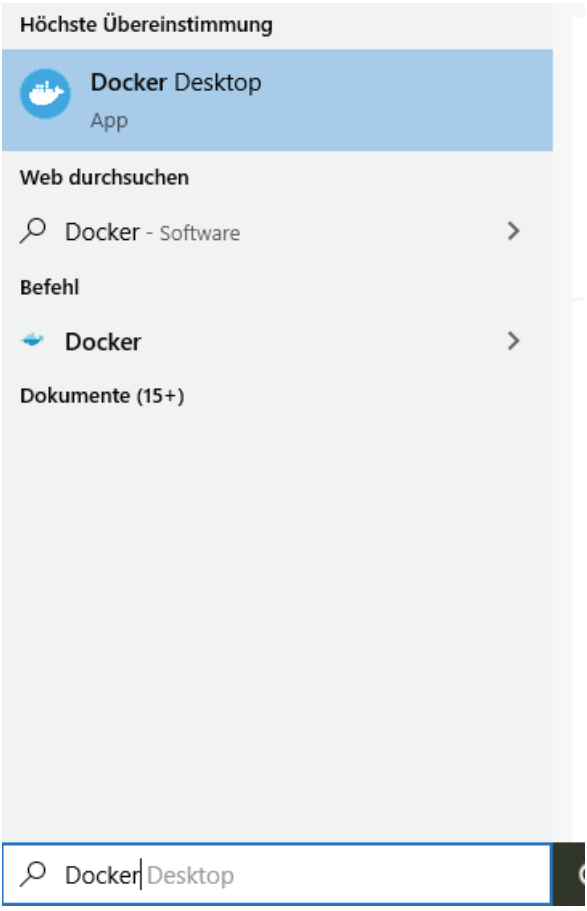



**Abb. 3.13** Start von Docker Desktop

Docker läuft jetzt als Hintergrund Dienst. Das Symbol, ein kleiner Wal, wird im Status Bereich der Windows-Taskleiste angezeigt. Bewegt man die Maus über das Symbol, wird der Status von Docker Desktop als Tool-Tip Text angezeigt (z.B. Docker Desktop is running, s. Abb. 3.14).

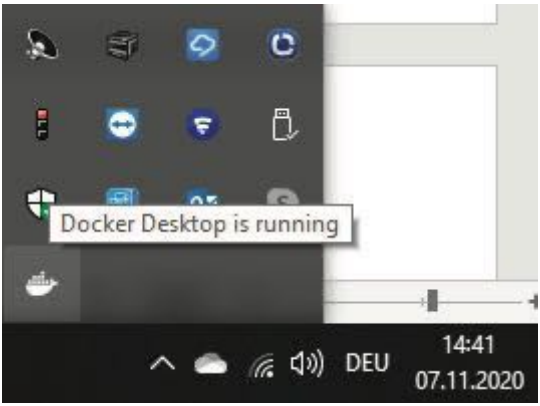



**Abb. 3.14** Docker-Symbol im Statusbereich der Windows-Taskleiste.

Klickt man mit der Maus auf das Symbol, erscheint ein Menü, über das Docker Desktop gesteuert werden kann. Hier kann man Docker-Einstellungen ändern oder auch Docker beenden (Abb. 3.15).

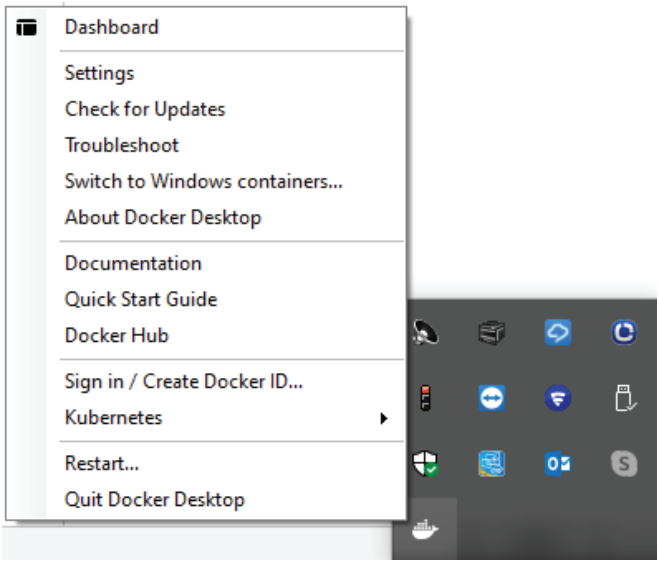



**Abb. 3.15** Das Docker Desktop Kontextmenü.

Über das Menükommando Switch to Windows Containers ... bzw. Switch to Linux Containers ... kann man den Containertyp ändern, den man während der Installation aus dem Installationsdialog ausgewählt hat.

Durch Mausklick auf den Menüpunkt Docker Hub öffnet man in einem Web-Browser die Docker Hub Webseite mit den Repositories. Mit Dokumentation wird man automatisch auf die Webseite mit der Docker Online-Dokumentation geleitet.



Die restlichen Menüpunkte werden in diesem Buch später in anderen Kapiteln genauer behandelt.

### 3.2.2 Docker Container starten

Für den Start eines Docker Containers nutzen wir die folgende Syntax:

```
1 docker run [<options>] <image_name>[:<tag>] '
2 [<command>] [<args>...]
```

Diese Syntaxbeschreibung ist nicht vollständig. Um eine detailliertere Beschreibung des `docker run`-Kommandos zu erhalten, schlagen Sie bitte im Anhang nach.

### 3.2.3 Beispiel-Image ‚Hello-world'

Starten Sie die PowerShell und geben Sie das folgende Docker-Kommando ein (Abb. 3.16):

```
1 > Docker run hello-world
```

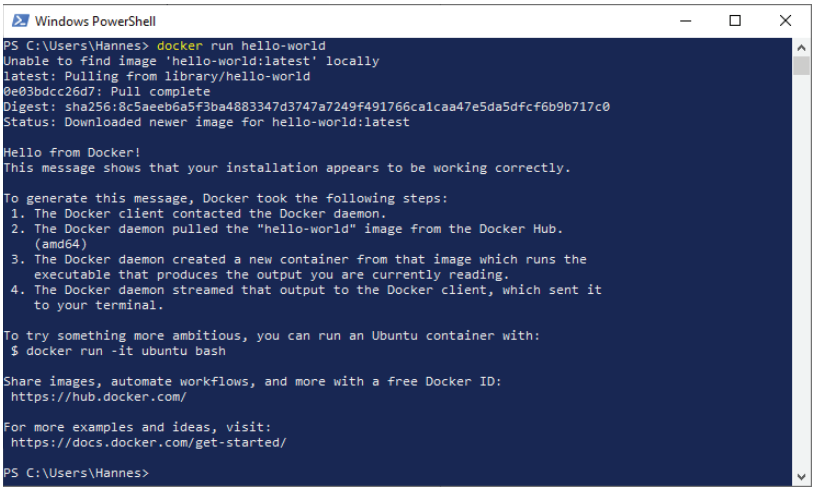



**Abb. 3.16** Run ‚hello-world' Container

Zuerst teilt Ihnen Docker mit seinen Ausgaben in der Shell mit, dass das Image von ‚hello-world' lokal nicht gefunden wurde und deshalb von ‚Docker Hub' heruntergeladen wird.

Danach kommen Informationen von der Docker-Anwendung ‚hello-world', welche Aktionen Docker mit diesem Kommando ausgeführt hat:

Der Docker Client hat sich mit dem Docker Daemon verbunden. Dieser hat das Image vom Docker Hub heruntergeladen. Der Docker Daemon hat aus dem Image einen neuen Container erstellt, der wiederum eine Anwendung ausführt, die den angezeigten Text ausgibt.

Mit dem Kommando

```
1  > docker image ls
```

bekommt man Informationen zu den aktuell heruntergeladenen Docker Images angezeigt (Abb. 3.17).

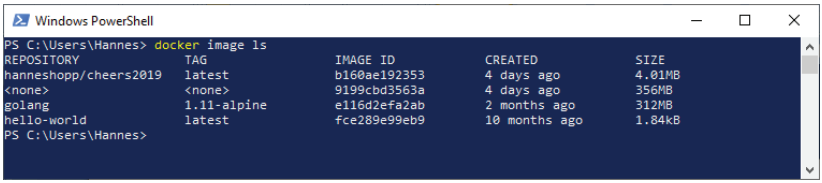



**Abb. 3.17** Ausgabe des Docker-Kommandos ‚image'



Damit haben wir jetzt schon die ersten beiden Docker-Kommandos kennengelernt: das Kommando `docker run` und das Kommando `docker image` mit dem Parameter `ls`.

Wenn Sie mehr Informationen über Docker-Kommandos erhalten wollen, dann geht das mit dem `help`-Kommando.

Hier das Kommando, um Hilfe zum run-Kommando zu erhalten:

```
1  > docker help run
```

Sie erhalten dann die folgende Ausgabe (Abb. 3.18 der Screenshot zeigt hier nicht alle Zeilen der Ausgabe).

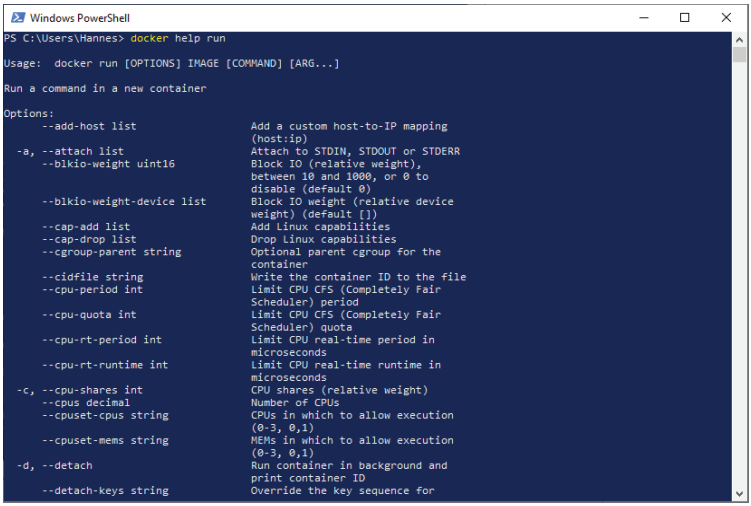



**Abb. 3.18** Ausgabe des Docker-Kommandos ‚help run'

Kapitel 4

# Docker-Grundlagen



## 4.1 Docker Hub nach Images durchsuchen

Als Nächstes wollen wir uns einen Überblick über die Docker Images verschaffen, die in Docker Hub verfügbar sind.

Dazu begeben wir uns auf die folgende Webseite:

https://hub.docker.com/

Sie können dazu auch aus dem Docker-Kontextmenü den Menüpunkt Docker Hub benutzen.

Zur Wiederholung – ein Klick mit der rechten Maustaste auf das Docker-Symbol im Statusbereich der Windows-Taskleiste öffnet das Docker-Kontextmenü.

Falls Sie nicht angemeldet sind, erscheint zuerst die Startseite von Docker Hub (Abb. 4.1):

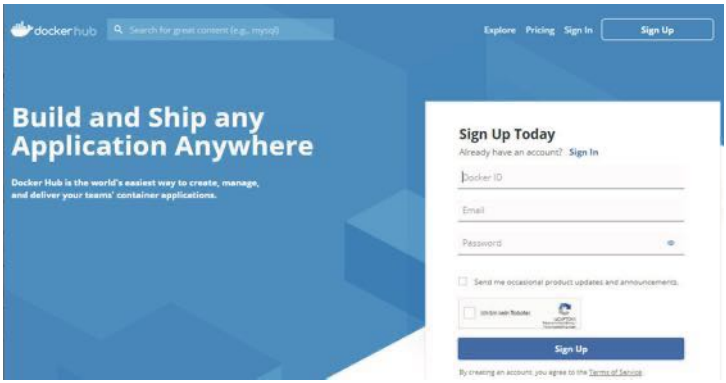



**Abb. 4.1** Die Startseite von Docker Hub

Klicken Sie hier mit der Maus auf den Menüpunkt SIGN IN oder alternativ auf den Button [SIGN UP FOR DOCKER HUB], um wieder auf die Anmeldeseite von Docker Hub zu gelangen.

Hier geben sie wieder Ihre Docker ID und das Passwort ein. Das sind die Informationen, die Sie vor dem Download der Installations-Dateien von Docker angegeben haben.

War der Login-Vorgang erfolgreich, dann sehen Sie jetzt die Repository-Seite von Docker Hub (Abb. 4.2).

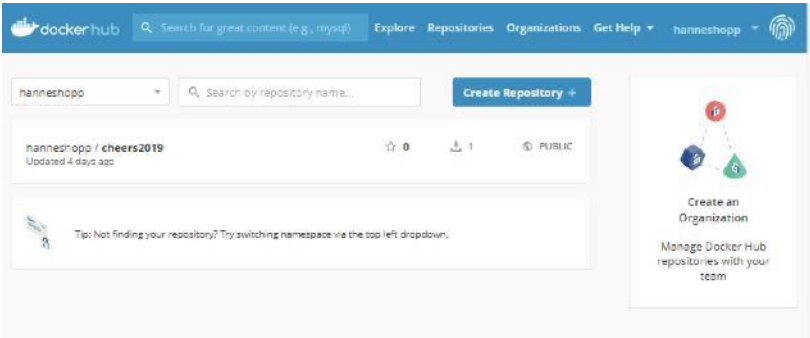



**Abb. 4.2** Die Repository-Seite von Docker Hub

Auf dieser Seite werden Ihre persönlichen Repositories aufgelistet. Wir kommen später wieder auf diese Seite zurück.

Unter dem Menüpunkt REPOSITORIES befinden sich Ihre eigenen Repositories. Im Moment wird da vermutlich noch nichts angezeigt. Das wird sich aber sehr bald ändern.

Um die Seite mit den offiziellen öffentlichen Repositories anzuzeigen, klicken wir jetzt auf den Menüpunkt EXPLORE.

Falls das Register CONTAINERS nicht bereits aktiviert ist, dann wählen Sie hier dieses.

Begeben Sie sich jetzt einmal ganz entspannt auf eine Entdeckungsreise durch die hier angebotenen Images (Abb. 4.3).

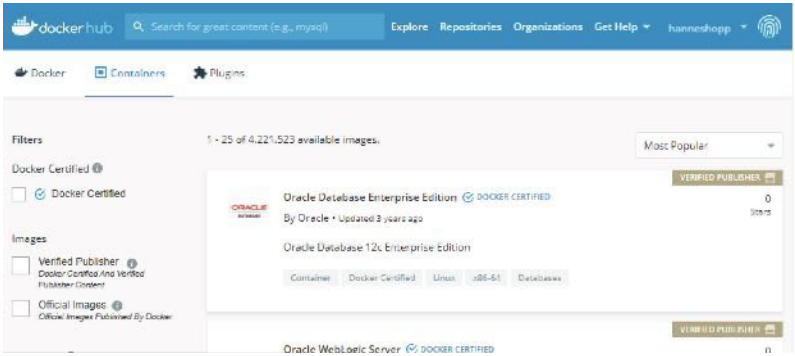



**Abb. 4.3** Die Container von Docker Hub

Versuchen Sie das Image von unserer eben getesteten ‚hello-world'-Anwendung zu finden und lesen Sie die dazugehörigen Informationen von Docker Hub.

Dazu scrollen Sie durch die Liste der angebotenen Images. Wenn Sie das Image „hello-world" gefunden haben, dann klicken Sie mit der Maus darauf, um mehr Informationen über diese Image zu erhalten.

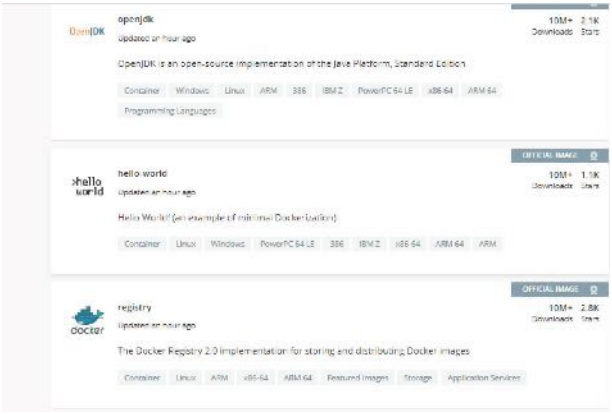



**Abb. 4.4** Das Image „hello-world" in Docker Hub

Für die Kommandozeilen-Fans unter Ihnen: Docker bietet auch ein CLI-Kommando, um den Docker Hub aus einer Shell heraus zu durchsuchen.

Starten Sie dazu unter Windows die PowerShell (oder unter Linux zum Beispiel bash) und geben Sie das folgende Kommando ein, um das „hello-world“ Image im Docker Hub zu finden:

```
1  > docker search hello-world
```

Überrascht?

Es erscheint gleich eine ganze Liste von Images, die den Text „hello-world“ im Namen tragen (Abb. 4.5).

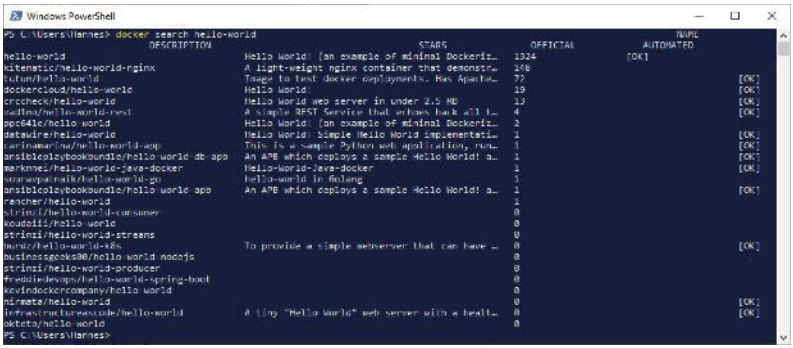



**Abb. 4.5** Docker-Kommando `search hello-world`

Das Docker `search`-Kommando liefert als Ausgabe die Namen der gefundenen Repositories, eine Kurzbeschreibung und zusätzlich sogar die Anzahl der Sterne, die durch Benutzer vergeben wurden.

In den beiden Spalten rechts wird angegeben, ob das Repository ein offizielles ist (OFFICIAL [OK]) und ob ein Repository automatisch erstellt wurde (AUTOMATED [OK]).

Auch Sie haben übrigens die Möglichkeit, ein Image, das Ihnen gefällt, mit einem Stern zu bewerten. Das erhöht dann auch die Anzahl der „STARS“ in der Ausgabe von `docker search`.

Um einen Stern zu vergeben, klickt man in der Container-Liste von Docker Hub auf das gewünschte Repository, um die Detail-Information dafür anzuzeigen. Oben rechts neben dem Image-Icon und dem Image-Namen wird ein kleiner Stern angezeigt. Mit einem Mausklick auf diesen Stern können Sie das Rating für das zugehörige Image erhöhen (Abb. 4.6).



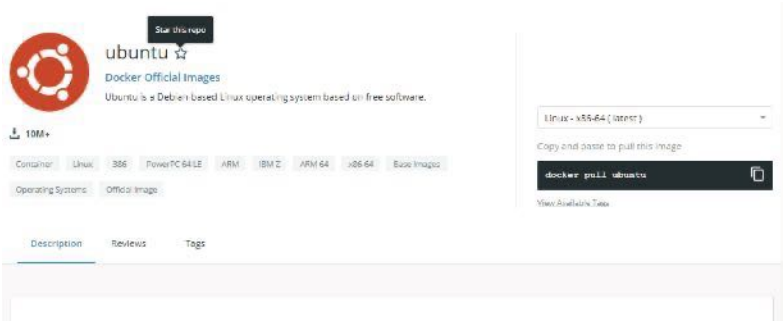



**Abb. 4.6** Ein Image mit einem Stern bewerten

## 4.2 Die Version eines Docker Images bestimmen

Bis jetzt haben wir beim Start unseres „hello-world“ Images nur den Image-Namen ohne Versions-Informationen angegeben.

```
1  > Docker run hello-world
```

Wird ein Image-Name bei Docker-Kommandos wie `docker run` oder `docker pull` ohne Versionsangabe ausgeführt, dann wird immer die neueste Version (:latest) des Images herangezogen.

Soll eine ältere Version von einem Image verwendet werden, um davon einen Container zu bauen, so kann das über ein *Tag* angegeben werden.

Im Docker Hub werden die *Tags* der verfügbaren Image-Versionen in den Detail-Informationen des Images angegeben.

Der folgende Screenshot zeigt die DESCRIPTION-Seite für das „ubuntu“ Image mit seinen Tags (Abb. 4.7).

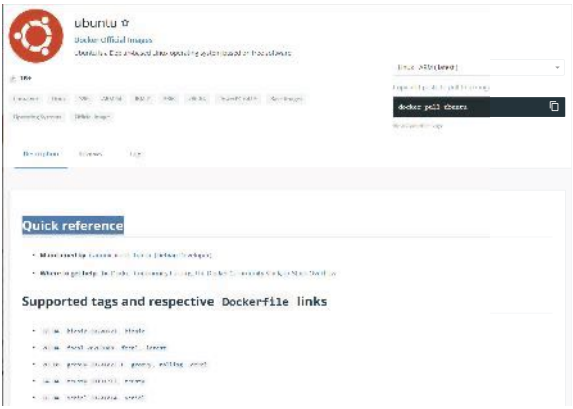



**Abb. 4.7** Tags-Beispiel: Ubuntu Image Tags

Wählt man auf dieser Seite das Register TAGS durch Mausklick, dann erhält man ausführliche Versions-Informationen über alle Tags (Abb. 4.8).

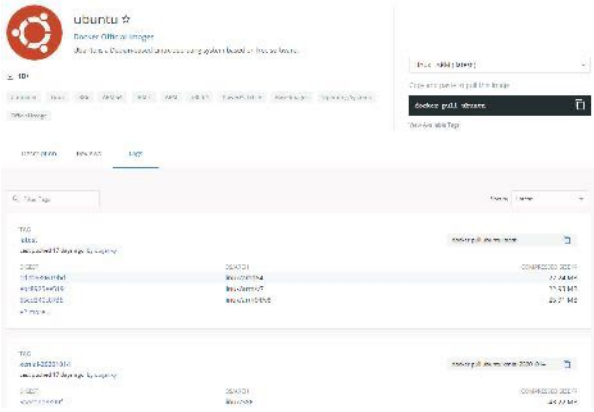



**Abb. 4.8** Register TAGS für das „ubuntu“ Image.

Beispiel: Container für ubuntu Xenial bauen.

Starten Sie die PowerShell und geben Sie dort das folgende Kommando ein:

```
1  > docker pull ubuntu:xenial
```

Docker holt sich jetzt die benötigten Bibliotheken vom Repository. Das kann eine Weile dauern.



Wurde das Kommando `docker pull` erfolgreich ausgeführt, erhalten Sie am Ende die folgende Anzeige (Abb. 4.9):

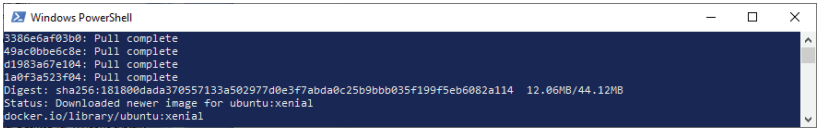



**Abb. 4.9** `docker pull` mit Versionsangabe

Wenn Sie danach das Kommando `docker image ls` ausführen, dann taucht das neu geladene Image in der ausgegebenen Liste an oberster Stelle auf. Die zweite Spalte dieser Liste zeigt das zum Repository gehörende TAG an. Hier sehen Sie, dass tatsächlich das Image für die Xenial-Version von Ubuntu heruntergeladen wurde (Abb. 4.10).

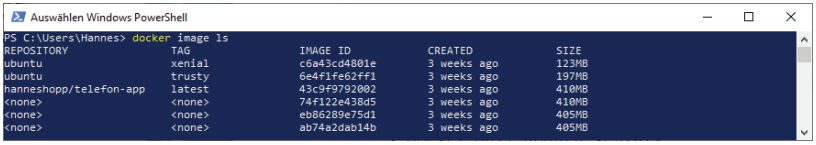



**Abb. 4.10** Anzeige der Image Liste mit ubuntu xenial

Wollen Sie das Image mit diesem Tag in einem Container ausführen, dann geben Sie mit dem `docker run` Befehl zum Image-Namen zusätzlich den Tag durch einen Doppelpunkt getrennt an.

```
1  > docker run -t -d --name my_xenial ubuntu:xenial
```

Es wird jetzt von Docker eine seltsame Zeichenfolge ausgegeben. Diese Zeichenfolge ist das sogenannte „Digest“ und dient der eindeutigen Identifizierung der Container.

Mit dem folgenden Kommando prüfen wir, ob der Container auch läuft (Abb. 4.11):

```
1  > docker container ls
```

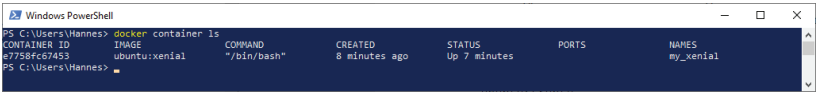



**Abb. 4.11** Anzeige des Container-Status

Bis jetzt sehen wir nicht viel von unserem Container, denn dieser läuft als Hintergrund-Prozess und produziert keine Ausgaben. Wir wollen jetzt aber in unseren Container ‚hineinsehen‘.

Für diesen Zweck gibt es das Kommando `docker exec`. Das erlaubt uns, in einem laufenden Container ein Kommando auszuführen. Wir starten mit diesem Kommando in unserem Container eine bash Shell und können darüber dann Linux-Kommandos eingeben und ausführen lassen (Abb. 4.12).

```
1  > docker exec -it my_xenial /bin/bash
```


**Abb. 4.12** bash Shell im ubuntu Xenial Container

Der Screenshot zeigt, wie von der PowerShell aus im Container eine bash Shell gestartet wird. Die Shell meldet sich mit dem Prompt `root@e7758fc67453:/#`

Innerhalb der Shell wird das Kommando `cat /etc/issue` ausgeführt. Das gibt in der nächsten Zeile die ubuntu Version aus (Ubuntu 16.04.6 LTS – wenn Sie im Internet recherchieren, werden Sie feststellen, dass dies eine „Xenial" Version ist).



Danach folgen die Linux-Befehle `whoami` und `ps`.

Versuchen Sie, selbst ein paar Linux Shell Kommandos einzugeben.

Beendet wird die Shell mit dem Kommando `exit`. Danach gehen Ihre Eingaben wieder an die Parent Shell, im obigen Beispiel an die Windows PowerShell.

## 4.3 Übungsaufgabe: Container für eine ältere Image-Version bauen

Um das bisher gelernte zu vertiefen, erhalten Sie hier eine kleine Übungsaufgabe.

Im Anschluss wird Ihnen ein Beispiel als mögliche Lösung vorgestellt. Natürlich gibt es nicht nur eine Lösung und wenn Ihr Lösungsansatz funktioniert, dann ist er logischerweise perfekt.

Hier Ihre Aufgabe:

Bauen Sie einen Container für „debian" Linux mit dem tag „jessie" und überprüfen Sie diesen.

***Lösung:***

```
1  > docker pull debian:jessie
2  > docker run -t -d --name my_debian debian:jessie
3  > docker exec -it my_debian /bin/bash
```

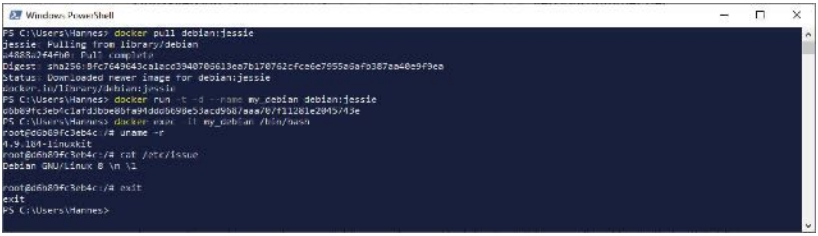



**Abb. 4.13** Übungsaufgabe Debian Jessie Container

## 4.4 Häufig verwendete Docker Images

In diesem Kapitel stellen wir Ihnen eine kleine Auswahl verschiedener Docker Images vor, welche im Allgemeinen sehr beliebt sind und häufig eingesetzt werden. Die Images werden hier nur kurz beschrieben. Detaillierte Informationen finden sie im Docker Hub selbst. Dort werden auch Links zu den Webseiten der Hersteller bereitgestellt.

Die Reihenfolge der in diesem Kapitel beschriebenen Docker Images ist alphabetisch und stellt damit weder eine Wertung dar, noch spiegelt sie die Reihenfolge in einer Beliebtheitsskala wider, da sich solche Einschätzungen bekanntlich recht schnell ändern können.

### 4.4.1 Couchbase

Couchbase

Der Couchbase Server, ursprünglich wurde er Membase genannt, ist eine NoSQL. Dies ist eine auf Dokumente orientierte Datenbank, die

optimal für interaktive Anwendungen eingesetzt werden kann. Couchbase zeichnet sich zudem durch seine hohe Skalierbarkeit aus und wird daher gerne in Clustern betrieben.

### 4.4.2 Arangodb



ArangoDB unterscheidet sich von anderen Datenbankbank-Systemen vor allem dadurch, dass es sich hier um eine „Multi-Model" Datenbank handelt. Die meisten Datenbanken sind um ein bestimmtes Datenmodell herum organisiert (Relationales Modell, Hierarchisches Modell, Dokumentenorientiertes Modell, ...). ArangoDB dagegen unterstützt drei Datenbank-Modelle gleichzeitig, nämlich das Key/Value Modell, ein dokumentenorientiertes Modell, und das Graph-Datenbankmodel. Es besteht aus einem Datenbank Kern und nutzt die Abfrage Sprache AQL (ArangoDB Query Language).

### 4.4.3 Apache http Server

Der Apache http Server ist eine Open Source Web-Server-Applikation der Apache Software Foundation.

Apache erlaubt es, dynamische Webseiten zu erstellen. Dabei ist der Einsatz von Server seitigen Script-Sprachen wie PHP, Perl oder Ruby möglich.

Apache ist Modul basiert. Apache Module können hier jederzeit aktiviert oder deaktiviert werden.

### 4.4.4 CentOS


CentOS steht für ***C**ommunity **Ent**erprise **O**perating **S**ystem*. Es handelt sich um eine Linux Distribution, die von einer Community aus freiwilligen Software Entwicklern entwickelt wurde und von dieser auch weiter gepflegt wird. CentOS basiert auf „Red Hat Enterprise Linux". CentOS steht mittlerweile an dritter Stelle der am häufigsten installierten Linux Distributionen, nach Ubuntu und Debian.

### 4.4.5 Elasticsearch


Bei Elasticsearch handelt es sich um eine der am weitesten verbreiteten Suchmaschinen. Diese Suchmaschine wird über ein REST Interface angesprochen. Die Applikation kann zahlreiche Dokumente in hoher Geschwindigkeit durchsuchen und analysieren.

Elasticsearch wurde in der Programmiersprache JAVA entwickelt. Dokumente und Suchanfragen (Queries) werden im JSON-Format ausgetauscht.

### 4.4.6 Fedora


Auch bei Fedora handelt es sich um eine Linux Distribution, welche von einer Online Community organisiert und von einem Gremium des Unternehmens „Red Hat" geführt wird.

Fedora ist der unmittelbare Nachfolger der „Red Hat“ Linux Distribution.

### 4.4.7 Jenkins




Jenkins ist eine Anwendung zur automatisierten kontinuierlichen Integration und kontinuierlichen Bereitstellung von Software Komponenten. Die Open-Source-Applikation ist webbasiert, leicht erweiterbar und konfigurierbar.

Jenkins ist wohl im Moment der meistbenutzte Open Source CI Server.

### 4.4.8 Joomla


Joomla ist ein kostenloses Open Source Content-Management-System für Webseiten. Es ist recht ähnlich zu WordPress und rangiert zurzeit in der Kategorie der CMS Systeme auf Platz zwei, nach WordPress.

Joomla wurde in der Programmiersprache PHP geschrieben und basiert auf einem „Model-View-Controller“ Framework für Web-Applikationen.

### 4.4.9 MariaDB

MariaDB ist ein relationales Datenbank Management System und bietet vergleichbare Funktionalität und Leistung wie MySQL. Bei MariaDB handelt es sich um eine Abspaltung von MySQL. So liegt MariaDB lediglich in der Beliebtheitsskala hinter MySQL.

### 4.4.10 MongoDB


MongoDB ist eine weitere Open-Source-Datenbank. Allerdings ist diese, im Gegensatz zu MySQL, dokumentenorientiert, also ein NoSQL-Datenbank-System.

MongoDB wurde in C++ entwickelt und verwaltet die Daten in einem Dokumentenformat, das JSON verwandt ist.

### 4.4.11 MySQL


Das weltweit wohl am meisten verbreitete Relationale Datenbank-Verwaltungssystem. In Docker Hub wird sogar behauptet, es sei die weltweit beliebteste Open-Source-Datenbank.

Mittlerweile ist MySQL führend beim Einsatz als Datenbank für Web-Applikationen, wie zum Beispiel Internet Shops.

Gerne wird MySQL zusammen mit dem Apache Server oder Nginx als Webserver eingesetzt. Als Skriptsprache wird dabei häufig PHP verwendet.

### 4.4.12 Neo4J


Neo4j ist die aktuell wohl populärste Open Source Graph-Datenbank. Bei einer Graph-Datenbank werden die Informationen nicht in Tabellen, sondern in ‚Graphen' gespeichert. Ein Graph ist dabei eine abstrakte Struktur, die Informationen über Objekte in einem System zusammen mit deren Beziehungen zu anderen Objekten speichert.

Als Programmiersprachen werden von Neo4j lediglich Java und Scala unterstützt.

### 4.4.13 Nginx


Nginx (man spricht das wie im Englischen Engine-X aus) ist ein Open Source Web Server, Reverse Proxy und E-Mail Proxy. Er unterstützt die Protokolle HTTP, HTTPS, SMTP, POP3, und IMAP.

Nginx kann darüber hinaus auch als Load Balancer eingesetzt werden. Damit können die Skalierbarkeit und die Zuverlässigkeit von Web-Applikationen verbessert werden.

Nginx ist im Moment das populärste Image im Docker Hub, auch wenn Apache immer noch der am weitesten verbreitete Webserver ist.

### 4.4.14 Node

Node.js ist eine Server seitige Plattform für Web-Applikationen. Node.js-Anwendungen werden in der Programmiersprache JavaScript entwickelt.

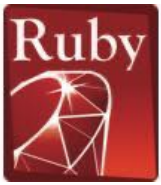
Diese Plattform zeichnet sich durch seine hohe Performance bei gleichzeitig niedrigem Bedarf an Arbeitsspeicher aus und unterstützt den Entwurf von nicht blockierenden Multi-Threading-Architekturen.

### 4.4.15 PostgresSQL


Hier ist noch eine weitere relationale Datenbank mit vergleichbarer Funktionalität zu MySQL und MariaDB. Es ist ein freies Objektrationales Datenbanksystem (ORDBMS).

Diesem Datenbanksystem wird unter anderem hohe Zuverlässigkeit, große Stabilität und Daten-Integrität zugesprochen.

### 4.4.16 Ruby






Ruby ist eine universelle Open-Source-Programmiersprache und zählt zu den höheren Programmiersprachen. Die Sprache ist dynamisch und objektorientiert, unterstützt aber auch Programmier-Paradigmen wie Funktionale Programmierung und Prozedurale Programmierung.

Ruby bietet darüber hinaus dynamische Typisierung, Reflexion und automatische Speicherverwaltung. Ruby-Programme werden zur Laufzeit interpretiert, also erst unmittelbar vor der Ausführung in Maschinensprache übersetzt.

### 4.4.17 SonarQube


SonarQube ist ein Open-Source Tool zur statischen Quellcode-Analyse. Es wird zur kontinuierlichen Quellcode-Inspektion innerhalb von Containern eingesetzt. Dadurch sollen sowohl die Code-Qualität verbessert werden als auch Risiken im Quellcode einer Anwendung rechtzeitig aufgedeckt werden.

### 4.4.18 Tomcat


Bei Tomcat (Englischer Begriff für Kater) handelt es sich um einen Open Source Webserver. Er realisiert die Java Servlet und JavaServer Pages Technologien. Tomcat dient bei ungefähr zwei Dritteln der Java Web-Applikationen als Host.

Docker Anwender können mit dem Tomcat Image Container erstellen, die als Applikations-Server für Java-basierte Anwendungen einsetzbar sind. Das erlaubt, diese Anwendungen lokal auszuführen und zu testen.

### 4.4.19 Ubuntu

Den meisten von Ihnen ist Ubuntu Linux sicher ein Begriff. Es handelt sich um eine Debian basierte Linux Distribution. Ubuntu hat sich zum Weltweit beliebtesten Linux Betriebssystem hochentwickelt und befindet sich derzeit jetzt auf Platz eins der Docker Platform Container.

Im folgenden Kapitel werden Sie Ihr erstes selbst gebautes Image auf der Basis von Ubuntu erstellen.

### 4.4.20 WordPress


Zurzeit ist WordPress eines der weltweit am meisten eingesetzten Content-Management-Systeme (CMS) zur Webseiten-Gestaltung. WordPress ist ein Open-Source-Produkt und ist frei sowie kostenlos verfügbar. Gegenwärtig sind über 30% aller Internet-Seiten mit WordPress entwickelt worden.



Der Einsatz von WordPress in Docker Containern erlaubt es Ihnen, Änderungen und Aktualisierungen im Code Ihrer Webseite durchzuführen und zu testen, ohne die aktuelle, „live“ geschaltete, Webseite zu beeinflussen.

In den Kapiteln mit den fortgeschrittenen Techniken werden wir Ihnen zeigen, wie man einen WordPress-Blog mit Docker Images aufsetzen und betreiben kann.

## 4.5 Ein „Hello Docker“ Image selbst gebaut

Es wird jetzt langsam Zeit, unser eigenes Docker Image zu entwickeln. Damit es nicht gleich zu Beginn zu komplex wird, erstellen wir zunächst noch eine einfache „Hello“-Anwendung – ich habe sie „Hello Docker“ genannt.

Das „Hello Docker“ Image, welches wir jetzt erstellen wollen, soll auf das Basis Image „Ubuntu“ aus dem Docker Hub aufbauen.

### 4.5.1 Ausführen und Test des „Ubuntu“ Images.

Damit wir sehen, was ein „Ubuntu“ Container alles kann, beginnen wir zunächst mit einem praktischen Test des „Ubuntu“ Basis Image.

Das geht erst einmal recht einfach. Wieder starten wir die PowerShell und geben dort das folgende Kommando ein:

```
1  > docker run -i -t ubuntu /bin/bash
```

Folgendes passiert jetzt im Hintergrund:

- Docker sieht nach, ob es das Image ‚Ubuntu‘ bereits lokal gibt. Wenn nicht, lädt Docker es von der Registry herunter (so wie wenn wir es manuell durch das Kommando `docker pull ubuntu` laden würden).
- Docker erzeugt einen neuen Container von dem heruntergeladenen Image (so wie wenn wir manuell das Kommando `docker container create` eingeben würden).
- Docker fügt dem Container als letzte Schicht ein Dateisystem mit Schreib- und Leserechten zu, damit der laufende Container lokal Verzeichnisse und Dateien erstellen und bearbeiten kann.
- Docker erzeugt eine Standard-Netzwerk-Schnittstelle (mit IP-Adresse). Mehr zum Thema Netzwerke folgt in einem späteren Kapitel.
- Docker startet den Container und führt `/bin/bash` aus. Durch den Parameter –i haben wir festgelegt, dass der Container interaktiv ausgeführt wird, und der Parameter `–t` bestimmt, dass Ein- und Ausgaben über das aktuelle Terminal (also PowerShell) geleitet werden.

Geben Sie jetzt zum Test ein paar Linux-Kommandos ein (Abb. 4.14).

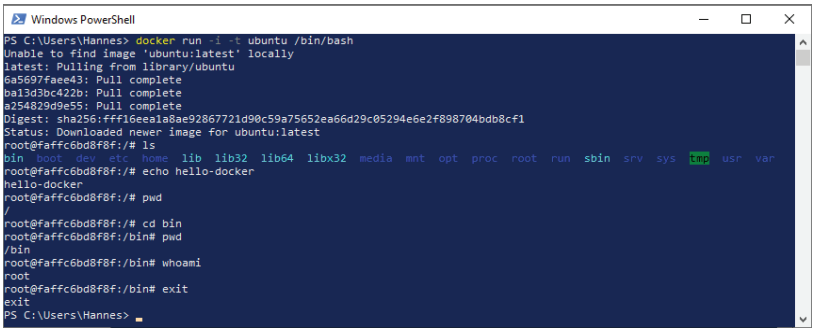



**Abb. 4.14** Linux-Kommandos im ‚Ubuntu‘ Container



### 4.5.2 Ein erstes einfaches abgeleitetes Image

Wir erzeugen unser eigenes, von ‚Ubuntu‘ abgeleitetes Image in einem eigenen Verzeichnis. Erstellen Sie dazu ein Verzeichnis mit dem Namen „`Hello-Docker`“ unter Ihrem Benutzerverzeichnis. Dieses Verzeichnis stellt den sogenannten „Build Context“ dar.

Unter dem Docker „Build Context“ versteht man einen Satz von Dateien, die sich in einem speziellen Verzeichnis befinden oder über eine spezielle URL erreicht werden können. Diese Dateien werden während des Build Vorgangs an die Docker Engine übertragen und stehen dann auf dem internen Dateisystem des Images zur Verfügung.

Wechseln Sie in das neu erstellte Verzeichnis „`Hello-Docker`“ und erstellen Sie dort eine Datei mit dem Dateinamen ‚Dockerfile‘.

Füllen Sie jetzt diese Datei mit der folgenden Docker-Anweisung:

```
1  Datei 'Dockerfile'
2  FROM ubuntu:latest
```

Mit diesem Eintrag geben wir an, dass das neue Image auf Basis des Images „Ubuntu“ aus dem Docker Hub erstellt werden soll. Nach dem Doppelpunkt wird ein Tag für die Version des Images angegeben. Der Einfachheit halber schreiben wir hier `latest` – das bewirkt, dass immer die neueste Version des Images als Basis für unser abgeleitetes Image herangezogen wird.

Wenn wir mit diesem Dockerfile ein Image erstellen, so erhalten wir eine exakte Kopie des ‚Ubuntu‘ Images mit einem neuen Image-Namen.

Wir erstellen jetzt das neue Image und starten dazu die PowerShell, wechseln in das Verzeichnis ‚Hello-Docker‘ und geben das folgende Kommando ein, um dieses Image zu erstellen.

```
1  > docker build -t <DOCKER_ID>/hello-docker .
```

Mit dem Kommandozeilen Parameter `-t` geben Sie an, dass eine Name und optional ein Tag im Format ‚name:tag‘ angegeben wird.

`<DOCKER_ID>` ist hier nur ein Platzhalter. Ersetzen Sie diesen durch die ID, welche Sie bei der Docker Registrierung angegeben haben (Abb. 4.15).

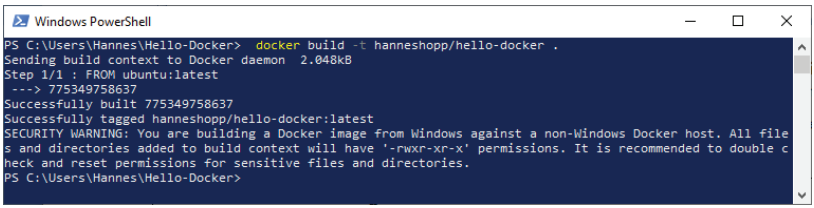



**Abb. 4.15** Kommando zum Bau des neuen ‚hello-docker‘ Images

Die Meldung sagt uns, dass das Image erfolgreich gebaut wurde. Da wir beim Aufruf kein Tag angegeben haben, wird als Vorgabewert ‚latest‘ verwendet.

Wir wollen nun sehen, ob das alles richtig geklappt hat, und führen das neue Image durch folgendes Kommando aus:

```
1  > docker run -i -t <DOCKER_ID>/hello-docker /bin/bash
```

ACHTUNG! Auch hier ist <DOCKER_ID> nur der Platzhalter für Ihre ID.



Nach dem Start des Kommandos in der PowerShell geben Sie als Test wieder verschiedene Linux-Kommandos ein.

Damit haben wir auf einfachste Weise unser erstes Docker Image erstellt und ausgeführt. Das hat im Moment aber noch keinen Vorteil zum Basis Image.

Wir werden unser neues Image also in den folgenden Schritten erweitern.

### 4.5.3 Erweiterung unseres Images

Bisher haben wir das `run`-Kommando mit dem Argument ‚`/bin/bash`‘ erweitert, damit ‚bash‘ als Kommando-Shell ausgeführt wird und wir über das Terminal mit der Anwendung interagieren können.

Wir erweitern als Nächstes unser Dockerfile um einen Eintrag, der dafür sorgt, dass dies ohne diesen Zusatz beim Start geschieht.

```
1  Datei 'Dockerfile'
2  FROM ubuntu:latest
3  CMD ["/bin/bash"]
```

Die Anweisung CMD bestimmt, welches Programm bei Containerstart ausgeführt wird. Es kann nur eine CMD-Anweisung im Dockerfile geben.

Bauen Sie das Image ‚Hello-Docker‘ neu. Starten Sie es anschließend ohne das letzte Argument `'/bin/bash'` und überprüfen Sie, ob Sie immer noch Shell-Kommandos ausführen können.

Im nächsten Schritt erweitern wir das Image so, dass ein Shell-Skript aufgerufen wird, welches den Text ‚Hello Docker' ausgibt.

Wir erstellen eine `bash` Skript-Datei mit folgendem Inhalt:

```
1  Datei 'hello.sh'
2  #!/bin/bash
3  echo hello-docker
```

Diese Datei speichern wir unter dem Namen ‚hello.sh' im gleichen Verzeichnis, in dem sich auch unser Dockerfile befindet.

**!!! ACHTUNG !!!**

Windows und Linux verwenden unterschiedliche EOL (End Of Line)-Sequenzen. Unter Windows ist das CR+LF (carriage return + line feed). Linux dagegen kennt nur LF als Zeilenende.

Wenn Sie einen Editor wie Notepad++ verwenden, können Sie einstellen, welche Option beim Abspeichern verwendet wird.

Bei Notepad finden Sie diese Auswahl unter dem Menüpunkt

Bearbeiten | Format | Zeilenende.

Ist hier nicht die Option ‚Konvertiere zu UNIX – LF' eingestellt, erkennt die Linux Shell die Kommandos aus diesen Skript-Dateien nicht richtig und gibt Fehlermeldungen aus. Auch im Dockerfile kann es u.U. zu Problemen dieser Art kommen.

Jetzt passen wir das Dockerfile so an, dass nicht mehr die Shell `bash`, sondern dieses Shell-Skript ausgeführt wird.

```
Datei 'Dockerfile'
FROM ubuntu:latest

COPY hello.sh .
RUN chmod +x ./hello.sh
CMD ["/hello.sh"]
```

Mit der COPY-Anweisung kopieren wir die Skript-Datei in das Dateisystem des Containers. Es wird dort im aktuellen Verzeichnis (.) angelegt. In diesem Fall ist es das `root`-Verzeichnis.



In Dockerfiles wird die Anweisung RUN verwendet, um ein beliebiges Programm zu starten.

In diesem Fall führt die RUN-Anweisung das `chmod`-Kommando aus, um für die Datei „hello.sh“ Execute-Rechte zu setzen. Damit ist dieses Skript als ausführbare Datei eingerichtet.

Zuletzt werden per CMD-Anweisung die Kommandos in der Datei ‚hello.sh‘ ausgeführt.

Wieder bauen wir unser Image neu:

```
> docker build -t <DOCKER_ID>/hello-docker .
```

Danach führen wir es wieder aus:

```
> docker run -i -t <DOCKER_ID>/hello-docker
```

### 4.5.4 Übungsaufgabe: Funktionalität des Images erweitern

Um das bisher Gelernte wieder zu vertiefen, erhalten Sie hier die zweite Übungsaufgabe:

Erweitern Sie das Shell-Skript so, dass die Frage ‚Whats your name?‘ ausgegeben wird und anschließend auf eine Tastatureingabe gewartet wird.

Nach der Tastatureingabe wird die Zeichenfolge

Hello <+ eingegebener Text>

im Terminal angezeigt.

Wenn das funktioniert, erweitern Sie das Skript so, dass beliebig oft hintereinander ein Name eingegeben werden kann und danach jeweils der Text ‚Hello <+ name>' angezeigt wird.

Erst die Eingabe von `'q'` soll die Ausführung des Containers beenden.

*Die folgenden bash Shell-Kommandos werden für diese Aufgabe benötigt:*

Kommentare beginnen mit dem Doppelkreuz (#) Zeichen. Alles, was rechts davon steht, wird ignoriert:

```
1  # das ist ein Kommentar
```

Ausgabe eines Textes:

```
1  echo <text>
```

Beispiel:

```
1  echo "Hello World"
```

Wartet auf eine Tastatureingabe und liest die Eingabe in eine Systemvariable:

```
1  read <variable>
```

Beispiel:

```
1  read EINGABE
```

Die `while`-Schleife:

```
1  while [bedingung]
2  do
3        <kommando>
```

```
        …
Done
```

Beispiel:

```
n=5
while [ $n -gt 0 ] # solange n größer als 0
do
   echo $n
   n=`expr $n - 1` # n ist gleich n minus 1
done
```



Bedingung für Zeichenketten:

```
<text_1> = <text_2> # True bei gleichen Zeichenketten
<text_1> != <text_2> # True bei ungleichen Zeichenketten
```

Wertzuweisung an Variablen:

```
<variable>=<wert>
```

Beispiel:

```
MESSAGE="Hello World"
```

Variable auslesen. Soll der Wert einer Variablen in einem Ausdruck verwendet werden, muss ein Dollarzeichen vor den Variablennamen gestellt werden:

```
$<variable>
```

Beispiel:

```
echo $message
```

***Lösung:***

```
Datei '<filename>'
#!/bin/bash
echo "What's your name?"
read ANSWER

while [ $ANSWER != 'q' ]
      do
            echo hello $ANSWER
            echo "What's your name? (or q to quit)"
            read ANSWER
done
```

Wieder bauen wir unser Image neu:

```
> docker build -t <DOCKER_ID>/hello-docker .
```

Danach führen wir es wieder aus:

```
> docker run -i -t <DOCKER_ID>/hello-docker
```

So sieht dann die Ausgabe in der Shell aus (Abb. 4.16):

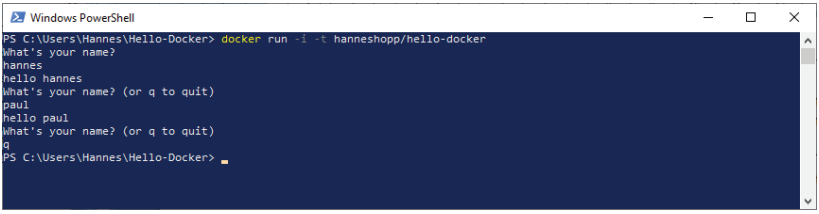



**Abb. 4.16** Übungsaufgabe: Ausgabe von „Hello-Docker“ in der Shell

## 4.6 Veröffentlichung des neuen Images in Docker Hub

Wir wollen jetzt endlich unser Docker Image im Docker Hub ablegen, damit zahlreiche andere Entwickler von unserer Arbeit profitieren können.

Dazu sind zwei Docker-Kommandos nötig:

```
> docker login
> docker push <DOCKER_ID>/hello-docker
```

Die Syntax des Push-Kommandos sieht wie folgt aus. Zusätzlich zum Image-Namen kann hier auch noch der Tag für die Version angegeben werden, die veröffentlicht werden soll (Abb. 4.17).

```
docker push [OPTIONS] NAME[:TAG]
```


**Abb. 4.17** Veröffentlichung des Images in Docker Hub

Tipp: Die beiden Kommandos können auch gleichzeitig in einer Zeile, durch ein Semikolon getrennt, eingegeben werden.

```
> docker login ; docker push <DOCKER_ID>/ hello-docker
```

Gehen sie jetzt im Internet auf die Webseite von Docker Hub

https://hub.docker.com/

und wählen Sie REPOSITORIES. Hier bekommen Sie das neue Image in einer Liste mit angezeigt (Abb. 4.18).

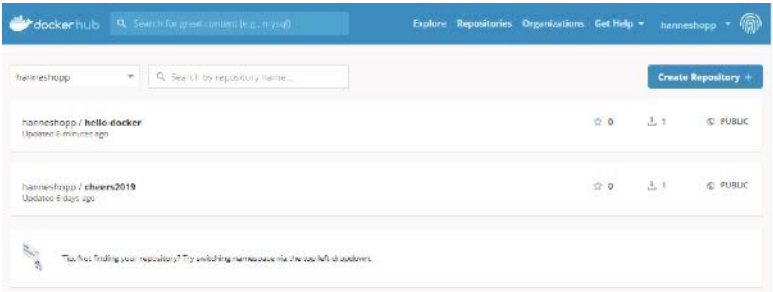



**Abb. 4.18** Repositories im Docker Hub

## 4.7 Docker Container im „detached"-Modus starten und stoppen

Bisher haben wir unsere Container im Vordergrund gestartet. Der Container wurde bei diesen Beispielen nach der Ausführung einiger Kommandos von selbst wieder beendet. Im Übungsbeispiel von „Hello-Docker" lief auch das Shell-Script in einer Schleife, die man durch Eingabe des Zeichens ‚q' beenden kann. Damit wird auch die Ausführung dieses Docker Containers beendet.

Normalerweise laufen aber in einem Container keine einfachen Skripten und auch User-Dialoge sind bei Web-Diensten nicht immer vorgesehen. Solche Anwendungen startet man üblicherweise im Hintergrund und die Programm-Anweisungen werden innerhalb einer Endlosschleife ausgeführt.

### 4.7.1 Container „detached" starten

Zunächst sehen wir uns an, wie ein Container im Hintergrund gestartet wird.

Man gibt dafür beim Container-Start den Parameter `-d` (detached) mit an (Abb. 4.19).

```
1  > docker run --name hello_docker -d hanneshopp/hello-docker
```

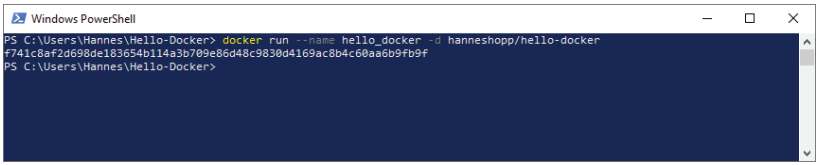



**Abb. 4.19** Container-Start in Detached Modus

Der Container läuft jetzt im Hintergrund und wir können im Kommando-Fenster weiterarbeiten. Natürlich ist jetzt so keine Kommunikation mit unserem Container mehr möglich und er kann jetzt auch nicht mehr durch die Eingabe des Zeichens ‚q' beendet werden.

Der Container muss also von außen gestoppt werden.

### 4.7.2 Container stoppen

Zum Stoppen eines Containers müssen wir das Docker-Kommando `stop` verwenden.

Die allgemeine Syntax zum `docker stop`-Kommando lautet:



```
1  > docker stop <container_name>
```

Für unser Beispiel sieht das Kommando dann so aus:

```
1  > docker stop hello_docker
```

Im Kommando-Fenster wird das erfolgreiche Stoppen des Containers durch die Ausgabe des Container-Namens bestätigt (Abb. 4.20).

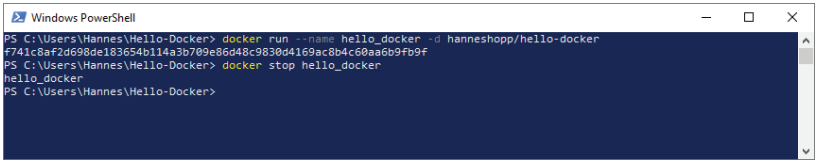



**Abb. 4.20** Stoppen eines Containers

### 4.7.3 Container wieder entfernen

Auch wenn die Ausführung des Containers beendet ist, existiert intern immer noch der Name des Containers mit seinem Container-Datei-System. Versucht man einen Container mit diesem Namen wieder zu starten, bekommt man eine Fehlermeldung (Abb. 4.21):

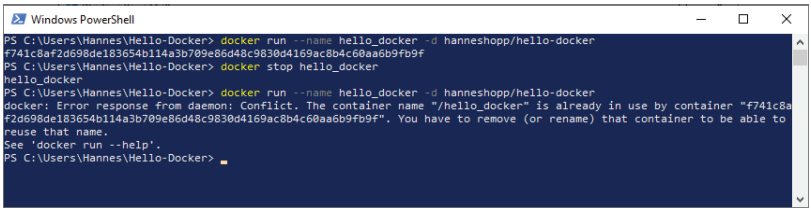



**Abb. 4.21** Fehler beim Start eines vorhandenen Containers

Man könnte jetzt beim Start einen neuen Container-Namen angeben. Besser ist es aber, den Container vor einem Neustart zu entfernen.

Die Syntax für dieses Kommando sieht folgendermaßen aus:

```
1 > docker container rm <container_name>
```

In unserem Beispiel lautet das Kommando:

```
1 > docker container rm hello_docker
```

Damit ist die Container-Instanz mit dem Namen `hello_docker` endgültig gelöscht und dieser Name kann beim Start eines neuen Containers wieder verwendet werden.

Wenn man sich den Aufwand ersparen will, dass man einen Container nach dem Stoppen zusätzlich noch aufräumen muss, dann kann man beim Container-Start durch die Angabe des Parameters `--rm` ein automatisches Cleanup beim Beenden des Containers bewirken.

ACHTUNG: Für Debug-Zwecke können Informationen aus dem Dateisystem des Containers durchaus hilfreich sein. Man sollte sich also gut überlegen, ob man das Löschen des Containers im operativen Betrieb automatisieren soll, da diese Daten dann auch verschwunden sind.

### 4.7.4 Container-Prozesse verwalten

In einem Docker Container werden Prozesse ausgeführt und verwaltet.

Der Container selbst ist auch ein Prozess, der seinerseits gestartet, gestoppt, gekillt und neu gestartet werden kann.

Die Kommandos `docker run` und `docker stop`, durch die ein Container gestartet bzw. gestoppt werden kann, haben wir in diesem Kapitel bereits kennengelernt.



#### *4.7.4.1 Anzeige der Containerliste*

Wenn Sie sehen möchten, welche Container gerade aktiv sind, können Sie sich eine Liste der laufenden Container mit dem Kommando `docker container ls ausgeben` lassen.

Sytax:

```
1  > docker container ls [<optionen>]
```

Beispiele:

Das folgende Kommando zeigt nur die gerade laufenden Container an (Abb. 4.22):

```
1  > docker container ls
```


**Abb. 4.22** Beispiel für das Kommando `docker container ls`

Wird als Parameter --all, oder die Kurzform -a zugefügt, zeigt die Ausgabe alle Container an, auch die beendeten (Abb. 4.23).

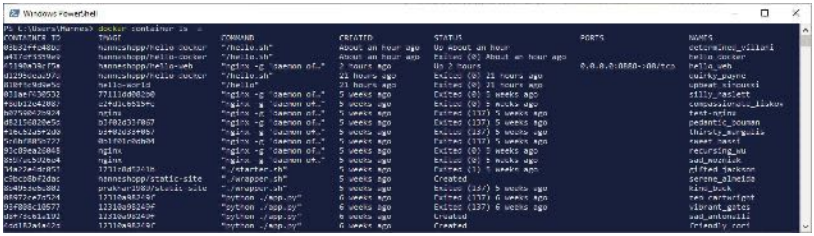


**Abb. 4.23** Beispiel für das Kommando `docker container ls` mit Parameter `-a`

Anstelle des Kommandos `docker container ls` kann auch das ältere Kommando

```
1 > docker ps
```

mit den gleichen Optionen verwendet werden. Es liefert das gleiche Ergebnis, die Verwendung wird aber mittlerweile von Docker nicht mehr empfohlen.

### *4.7.4.2 Container „Killen“*

Wenn das `stop`-Kommando allerdings aus irgendwelchen Gründen nicht funktioniert, gibt es noch eine „harte“ Version, einen Container-Prozess zu beenden: das Kommando `docker kill`. Dieses Kommando funktioniert, so wie Sie das vielleicht von LINUX oder anderen UNIX-Betriebssystemen her kennen. Man kann damit einen oder auch mehrere Container-Prozesse „killen“.

Hier erst zunächst die allgemeine Syntax für das Kommando:

```
1 > docker kill [<optionen>] <container> [<container> …]
```

Um den gewünschten Container anzugeben, kann sowohl der Name eines laufenden Containers angeben werden (nicht zu verwechseln mit dem Image-Namen) als auch die Container ID (oder auch Container Digest) oder das ID-Prefix (das sind die ersten Zeichen der Container ID oder des Container Digest).

Verwechseln Sie einen Container-Namen nicht mit dem Image-Namen. Der Container-Name kann beim docker `run`-Kommando optional über den Parameter `--name` übergeben werden. Lässt man diesen Parameter weg, vergibt die Docker Engine beim Start selbst einen Namen. Der aktuell verwendete Container-Name wird vom Kommando `docker container ls` bei der Ausgabe in der Spalte NAMES angezeigt.

Beispiel:



Der folgende Screenshot zeigt, wie zwei laufende Container nacheinander mit dem Kommando `docker kill` beendet werden (Abb. 4.24).


**Abb. 4.24** Beispiel zum Aufruf des Kommandos `docker kill`

Der erste Aufruf im obigen Beispiel gibt den Namen des Containers, der beendet werden soll, als Parameter an. Im zweiten Beispiel kommt die Variante mit dem ID-Prefix zur Identifizierung des gewünschten Containers zum Einsatz.

### *4.7.4.3 Anzeigen der internen Container-Prozesse*

Innerhalb eines Containers läuft mindestens ein Prozess. Es können aber auch mehrere Prozesse aktiv sein.

Um eine Liste der laufenden Prozesse eines speziellen Containers zu erhalten, verwenden wir das CLI-Kommando `docker top`.

Hier die Syntax

```
1  > docker top <container>
```

Um den gewünschten Container anzugeben, können auch hier wieder der Name eines laufenden Containers angeben werden, die Container ID oder das ID-Prefix.

Beispiel:

Der folgende Screenshot zeigt zwei Varianten, wie für einen laufenden Container die Informationen über seine Internen Prozesse abgefragt werden können (Abb. 4.25).


**Abb. 4.25** Das CLI-Kommando ‚`docker top`'

Das erste Beispiel verwendet den Namen des Containers (`hello_web`), der als Parameter abgefragt werden soll. Das zweite Beispiel nutzt das ID-Prefix zur Identifizierung des gewünschten Containers.

## 4.8 Eine einfache Webseite mit NGINX Image

### 4.8.1 Ausführen und Test des ‚NGINX' Images.

Wieder beginnen wir zunächst mit einem ersten Test des ‚NGINX' Images.

Diesmal starten wir in der PowerShell das folgende Kommando:

```
1 > docker run --name test-nginx -d -p 8080:80 nginx
```

Mit diesem Kommando starten wir das Basis Image von NGINX unter dem Container-Namen ‚testnginx'. Durch den Parameter –d geben wir an, dass der Container im Hintergrund ausgeführt wird (detached). Um den internen Port 80 auf dem externen Port 8080 zu veröffentlichen, wird der Parameter –p 8080:80 angegeben (publish).

Um zu kontrollieren, ob der Container wirklich läuft, geben wir dieses Kommando ein:



```
1  > docker container ls
```

Es wird dadurch eine Liste der aktiven Container ausgegeben. Wenn der Container mit dem Namen test-ngnx angezeigt wird, war das `run`-Kommando erfolgreich (Abb. 4.26).

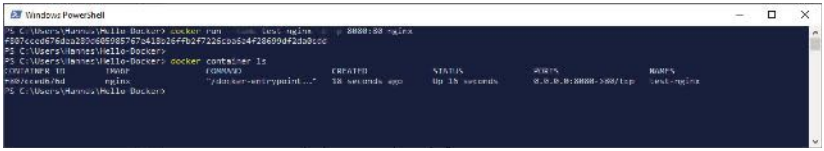



**Abb. 4.26** Das Docker-Kommando ‚container ls'

Zu sehen ist allerdings noch nichts von unserer Container-Anwendung. Dazu müssen wir in einem Webbrowser die Verbindung zu diesem Service herstellen.

Starten Sie Ihren Lieblings Webbrowser, zum Beispiel Google Chrome, Firefox, Microsoft Edge oder irgend einen anderen. Dann geben Sie dort in der Adressleiste die folgende Adresse ein:

http://localhost:8080/

Es erscheint die Webseite unseres Containers (Abb. 4.27).

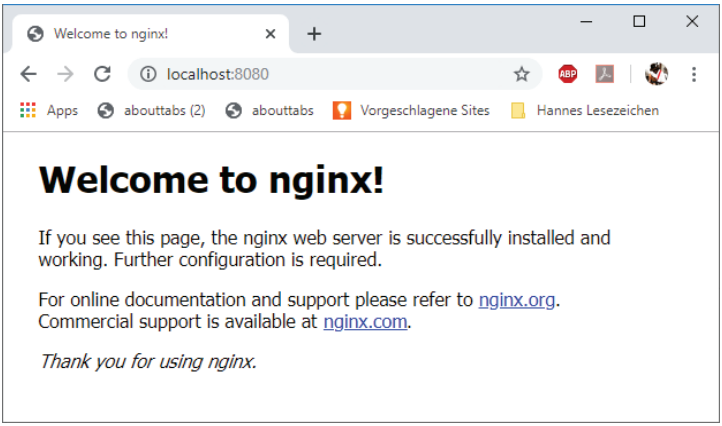



**Abb. 4.27** Die Webseite des NGINX Containers

### 4.8.2 Unsere eigene Webseite mit NGINX

Wir erzeugen wieder ein eigenes, diesmal von ‚NGINX ' abgeleitetes Image.

Erstellen Sie dazu ein Verzeichnis mit dem Namen ‚Hello-Web' unter Ihrem Benutzerverzeichnis und wechseln Sie in dieses Verzeichnis.

Erstellen Sie dort ein Unterverzeichnis mit dem Namen `html`.

Dort legen wir eine einfache HTML-Datei für unsere eigene Webseite an. Geben Sie dieser den Dateinamen `index.html`. Wie Sie wahrscheinlich wissen, dienen Dateien mit dem Namen ‚index.html' als Startseite eines Internet-Auftritts.

Füllen Sie die Datei `index.html` mit dem folgenden Inhalt (als Autor können Sie natürlich Ihren eigenen Namen angeben):

```
Datei 'index.html'
<!DOCTYPE html>
<html lang="en">
<head>
      <meta http-equiv="content-type" content="text/html;
```

```
      charset=utf-8">
      <meta http-equiv="content-type" content="text/html;
      charset=utf-8"/>
      <meta name="description" content="Eine einfach Webseite
      für Nginx"/>
      <meta name="author" content="Hans-M. Hopp"/>
      <meta name="keywords" content="Docker, Handbuch, Nginx,
      Hello, "/>
      <meta name="date" content="2019-12-05"/>
  <!-- Title-->
  <title> Hello Web </title>
</head>
<body>
  <div class="container">
    <div class="row" style="margin-top: 10%; margin-left: 10%">
          <h1 style="color:red">Hello Web!</h1>
          <p>Diese Seite wird in einem <strong>docker</strong>
          container mit Nginx ausgeführt.</p>
    </div>
  </div>
</body>
</html>
```

Jetzt benötigen wir noch ein neues Dockerfile. Das muss sich wieder in dem Verzeichnis befinden, das den „Build Context“ repräsentiert, also in unserem Fall das Verzeichnis `Hello-Web`. Erstellen Sie dort ein neues Dockerfile mit folgendem Inhalt:

```
Datei 'Dockerfile'
FROM nginx:latest
COPY html /usr/share/nginx/html
```

Das neue Image wird hier auf Basis des Images „Nginx“ aus dem Docker Hub erstellt. Auch hier wird durch das Tag `latest` bestimmt, dass die neueste Version als Basis Image zum Einsatz kommt.

Das Verzeichnis mit dem Namen `'html'`, in dem sich unsere Datei ‚index.html‘ befindet, wird im Container in das Verzeichnis `'/usr/share/nginx/html'` kopiert.

Mit dieser Konfiguration bauen wir ein neues Image. Wir starten die PowerShell, wechseln in das Verzeichnis ‚Hello-Web‘ und geben das folgende Kommando ein:

```
> docker build -t <DOCKER_ID>/hello-web .
```

Bitte vergessen Sie bei dem obigen Kommando nicht den Punkt am Ende. Damit geben wir an, dass sich der Build Context im aktuellen Verzeichnis befindet.

War der Build erfolgreich, dann starten wir unser neues Image als Container:

```
> docker run --name hello_web -d -p 8888:80 '
<DOCKER_ID>/hello-web
```

Und so sieht die Webseite im Browser aus (Abb. 4.28):


**Abb. 4.28** Unsere Webseite ‚Hello-Web'

Zuletzt wollen wir den laufenden Container wieder stoppen:

```
> docker stop hello_web
```

Sicherheitshalber sehen wir noch nach, ob der Container wirklich beendet wurde:

```
> docker container ls
```

Unsere Webseite sollte im Browser nicht mehr erreicht werden (Fehlermeldung im Browser). Falls sie doch noch angezeigt wird, liegt sie noch im Puffer des Browsers.

Wenn Sie aber den Befehl Aktualisieren (Neu Laden) ausführen, sollte der Browser die Fehlermeldung ausgeben.

Docker entfernt einen Container nicht automatisch, wenn er gestoppt wird. Ein erneuter Aufruf des Kommandos `docker run` führt daher zu einer Fehlermeldung (Abb. 4.29):



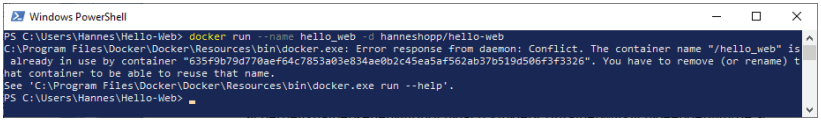



**Abb. 4.29** Fehlermeldung beim Kommando `docker run`

Unser Container wird mit dem folgenden Kommando entfernt:

```
1  > docker container rm hello_web
```

Sie können den Container auch umbenennen, so wie es das folgende Beispiel zeigt:

```
1  > docker container rename hello_web hello_docker
```

## 4.9 Eine etwas aufwendigere Webseite mit dem PHP Image

Um eine Einführung zum Thema Docker Container mit PHP zu geben, erzeugen wir noch ein einfaches, diesmal von ‚PHP' abgeleitetes Image.

Das Verzeichnis für unseren neuen Build Context, in dem die Image Daten angelegt werden, erhält den Namen `'Telefon-PHP'` und wird wieder unter Ihrem Benutzerverzeichnis eingefügt. Darin benötigen wir ein Unterverzeichnis, das den Namen `'src'` erhält. Dort soll jetzt eine Datei mit dem PHP-Skript angelegt werden.

Wechseln Sie in dieses Verzeichnis (<user_home>/Telefon-PHP/src).

In diesem Verzeichnis erstellen wir eine einfache PHP Script-Datei. Eine PHP Script-Datei enthält sowohl die HTML Tags der Webseite als auch den auszuführenden PHP Code der innerhalb eines eigenen Tags in das HTML-Dokument eingebettet ist.

Die neue Webseite, die wir jetzt erstellen wollen, besteht aus einem Formular mit einem Eingabefeld und einem Butten. Hier soll es möglich sein, dass ein Anwender einen Namen eingibt und anschließend auf einen Button mit dem Namen [SUCHEN] klickt. Ist der eingegebene Name bekannt, wird dieser mit einer zugehörigen Telefonnummer ausgegeben.

Beginnen wir mit einer leeren Datei, die den Dateinamen ‚**index.php**' erhält.

Diese Datei füllen wir mit dem folgenden Inhalt:

```
Datei 'index.php'
<!DOCTYPE html>
<html lang="de">
<head>
      <title> Telefon Liste PHP </title>
</head>

<body bgcolor="lightgreen">
<h1>Telefonnummer Suche</h1>

<!-- Eingebeddeter PHP Code -->
<?php
if(!isset($_GET['surname']))
{
      $currentName= "";
}
else
{
      $currentName = $_GET['surname'];
}

if($currentName != "")
{
      if(checkName($currentName))
      {
            echo "Der Name ". $currentName . " wurde gefunden";
```

```
27        }
28        else
29        {
30              echo "Unbekannter Name: ". $currentName;
31        }
32  }
33
34  function checkName($userName)
35  {
36        $userList = array(    "Hannes" =>           "089/73227",
37                                    "Heidi" =>      "089/73226",
38                                    "Philipp" =>    "099/5755",
39                                    "Paul" =>       "099/12345"
40                              );
41
42        if (array_key_exists($userName, $userList))
43        {
44              echo "<p>Name: $userName, Tel: $userList[$userName]</
45              p>";
46              return true;
47        }
48        return false;
49  }
50  ?>
51
52  <!-- Formular Bereich -->
53  <form action="index.php" method="get">
54
55  <p>Geben Sie einen Namen ein:
56  <input type="text" name="surname">
57  </p>
58
59  <p>
60  <input type="submit" value="Suchen">
61  </p>
62
63  </form>
64
65  </body>
```

Kernthema in diesem Buch ist zwar nicht PHP, sondern Docker. Trotzdem möchte ich hier eine kurze Beschreibung des PHP-Skripts abgeben.

Diese Skript-Datei besteht aus mehreren Bereichen.

Das ist, wie in einer HTML-Datei, der Header Bereich `<head>` ... `</head>`. Der unterscheidet sich nicht vom Header Bereich in HTML-Dateien. Um Platz zu sparen, wurden hier die Meta-Angaben weggelassen, sollten aber von Ihnen in Ihrer Version mit eingetragen werden (siehe die Datei **‚index.html'** im NGINX Beispiel).

Auch in dieser Datei gibt es den Body-Bereich `<body>` ... `</body>`. Wir haben mit der Angabe `bgcolor="lightgreen"` für den gesamten Body-Bereich als Hintergrundfarbe Hellgrün eingestellt.

Der Body-Bereich besteht wiederum aus mehreren Abschnitten.

Als Erstes wird auf unserer Webseite die Überschrift `'Telefonnummer Suche'` angezeigt.

Danach kommt ein Bereich mit eingebettetem PHP Code. Wir prüfen hier als Erstes, ob im Eingabefeld ‚surname' (im Formularbereich) ein Text eingegeben wurde. Wenn dort etwas steht, wird eine Funktion mit dem Namen `checkName()` aufgerufen. Diese bekommt den eingegebenen Text als Parameter übergeben.

Die Funktion `checkName()` durchsucht ein lokal definiertes Array nach der Zeichenfolge, die mit dem Parameter `'$userName'` übergeben wird. Wird der Name gefunden, wird dieser mit der zugehörigen Telefonnummer angezeigt und der Returnwert ‚true' zurückgegeben. Ist der Name nicht in dem Array vorhanden, gibt die Funktion ‚false' zurück.

Am Ende befindet sich der Formularbereich. Hier wird ein Text-Eingabefeld ‚surname' definiert, das die Eingabe eines Namens ermöglichen soll. Dazu kommt ein Button mit dem Label ‚Suchen'. Die ‚action' Angabe bestimmt für dieses Formular, dass sich das Skript selbst wieder aufruft, wenn der Button geklickt wird.

Jetzt erstellen Sie im Verzeichnis darüber (**<user_home>/Telefon-PHP**) das Dockerfile mit folgendem Inhalt:

```
1  Datei 'Dockerfile'
2  FROM php:7.2-apache
```

```
3 COPY src/ /var/www/html/
```

Das neue Image wird hier auf Basis des Images ‚php:7.2' in der Variante ‚apache' aus dem Docker Hub erstellt.

Das Verzeichnis mit dem Namen ‚src', in dem sich unsere Datei ‚index.php' befindet, wird im Image in das Verzeichnis '/var/www/html/' kopiert.



Mit dieser Konfiguration bauen wir das neue Image. Wir starten die PowerShell, wechseln in das Verzeichnis , Telefon-PHP ' und geben das folgende Kommando ein (wieder den Punkt am Ende nicht vergessen):

```
1 > docker build -t <DOCKER_ID>/telefon-app .
```

War der Build erfolgreich, dann starten wir einen Container auf Basis des neuen Images mit dem folgenden Kommando:

```
1 > docker run --name telefon_app -d '
2 -p 8808:80 <DOCKER_ID>/telefon-app
```

Und so sieht sie aus, die PHP-Webseite (Abb. 4.30):

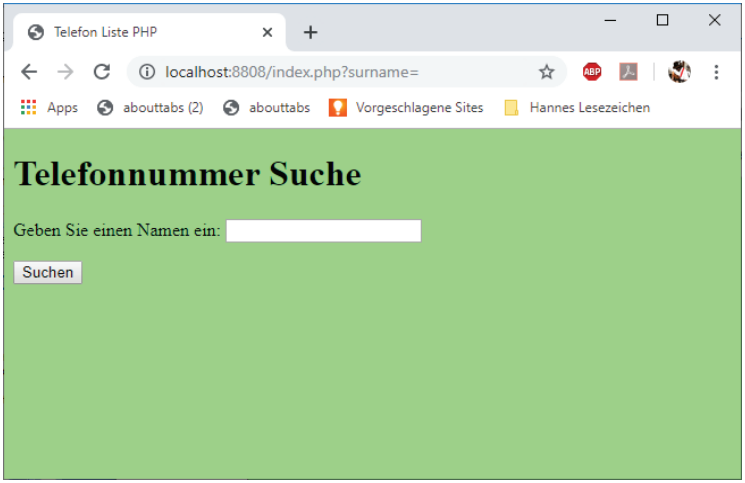



**Abb. 4.30** Webseite mit Telefonnummern-Suche

Geben wir hier den Namen ‚Paul' ein und klicken auf den Button [Suchen], dann verändert sich die Anzeige wie folgt (Abb. 4.31):

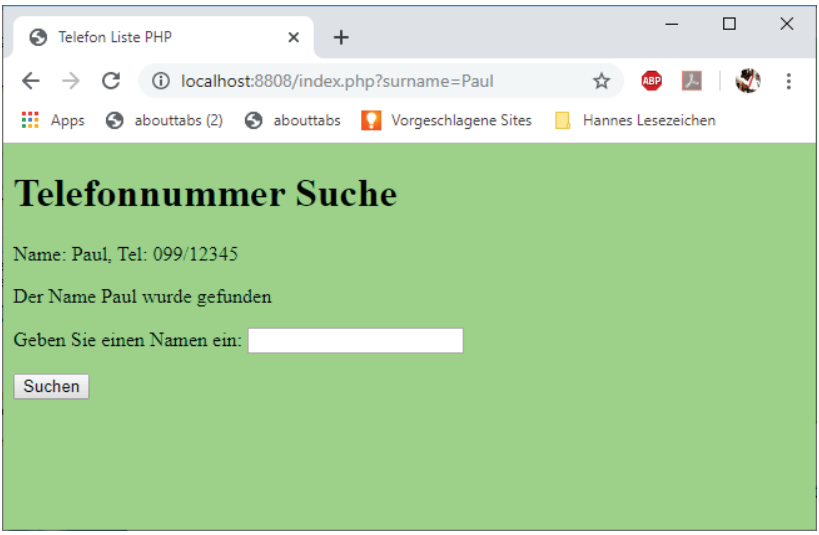



**Abb. 4.31** Telefonnummer-Suche – mit Ergebnis

Sie werden jetzt zu Recht bemängeln, dass eine Telefonliste, die im Code hart codiert ist, nicht besonders praktisch ist. Wir werden aber später in diesem Buch das Beispiel so erweitern, dass für die Verwaltung der Daten eine Datenbank verwendet wird.

Zuletzt veröffentlichen wir das neue Image in Docker Hub.

Zuerst muss man sich wieder einloggen:

```
1 > docker login
```

Anschließend wird das Image veröffentlicht:

```
1 > docker push <DOCKER_ID>/telefon-php
```

Kapitel 5

# Tools zur Arbeit mit Docker

## 5.1 Einfache Editoren

Grundsätzlich kann man mit jedem ASCII Editor arbeiten, wie z.B. dem Windows Editor, gedit oder auch der berühmte editor vi unter UNIX-Betriebssystemen.

Mittlerweile gibt es aber eine ganze Reihe von Code Editoren, die kostenlos im Internet zur Verfügung gestellt werden und die deutlich mehr Komfort für die Bearbeitung von Codetexten bieten.

Da gibt es Funktionen wie Syntax-Hervorhebung, Auto-Vervollständigung, automatische Einzüge, Multi-Dokument und Mehrfach-Fenster, erweiterte Suchen / Ersetzen Funktionen, Klammerhervorhebung, Lesezeichen, Makro-Aufzeichnung und Wiedergabe, Rechtschreibprüfung und vieles mehr.

An dieser Stelle soll nur eine kleine Auswahl der möglichen Text Tools vorgestellt werden. Es gibt aber noch viele andere und die Auswahl hängt sicher auch von persönlichen Vorlieben ab. Wie wir schon gesehen haben, ist es allerdings wichtig, dass man das Zeilenende-Format bestimmen kann (CR/LF, CR, LF).

- notepad++
  (https://notepad-plus-plus.org/)
- Atom
  (https://atom.io/)
- Emacs
  (https://www.gnu.org/software/emacs/)

## 5.2 Visual Studio Code und Docker CLI

Bei Visual Studio Code handelt es sich um einen leistungsfähigen Quellcode Editor, der sowohl für Windows, für macOS als auch für Linux verfügbar ist. Die Unterstützung für JavaScript, TypeScript, und node.js ist bereits integriert. Für weitere Sprachen (C++, C#, Java, Python, PHP, Go) sind zahlreiche Erweiterungen verfügbar. Es werden auch unterschiedliche Laufzeitumgebungen wie .NET und Unitiy unterstützt.

Zusammen mit der Installation von *Docker für Mac* oder *Docker für Window* können Sie eine einzelne Docker-CLI verwenden, um Anwendungen sowohl für Windows als auch für Linux zu erstellen. Außerdem unterstützt Visual Studio Code mit IntelliSense für Docker-Dateien und Verknüpfungsaufgaben, um Docker-Befehle aus dem Editor auszuführen.



Visual Studio Code kann im Internet von der folgenden Webseite für verschiedene Windows-Versionen, für verschiedene Linux-Distributionen und auch für macOS heruntergeladen werden:

https://code.visualstudio.com/Download

### 5.2.1 Visual Studio Remote WSL

Die Remote - WSL Erweiterung unterstützt den Einsatz des Windows-Subsystems für Linux (WSL) als Entwicklungsumgebung, die direkt aus dem Visual Studio Code Editor heraus genutzt werden kann.

Diese Erweiterung unterstützt den Entwickler:

- bei der Entwicklung in Linux basierter Umgebung unter Einbeziehung von Linux spezifischen Entwicklungswerkzeugen.
- beim Editieren von Dateien, die sich in WSL oder in einem eingehängten (mounted) Windows-Dateisystem (z.B. `/mnt/c`) befinden.
- in VS Code wird eine Linux basierte Anwendung direkt unter Windows ausgeführt und ge-debugt.

Kommandos und Erweiterungen werden direkt in WLS ausgeführt. Das bedeutet, dass Probleme durch Unterschiede in den beiden Betriebssystemen vermieden werden. VS Code wird in WSL genauso gehandhabt wie unter Windows.

### 5.2.2 Microsoft Docker Erweiterungen für VS Code

VS Code bietet zusätzlich Erweiterungen für die Arbeit mit Docker an. Es erleichtert und unterstützt die Entwicklung und die Bearbeitung von Dockerfiles.

Wenn Sie mit Docker Compose arbeiten, unterstützt Sie Visual Studio Code auch bei der Entwicklung und Bearbeitung der Docker Compose Konfigurationsdateien. Das sind ‚YAML' Dateien und tragen die Dateinamen `docker-compose.yml` und `docker-compose.debug.yml`.

Visual Studio Code bietet sogar die Möglichkeit an, Docker-Dateien automatisch korrekt zu generieren, so dass diese zum aktuellen Projekt-Typ passen.

Mehr zum Thema Docker Compose erfahren Sie später im Kapitel ‚Container betreiben mit Docker Compose'.

Die Installation der Docker-Erweiterungen für Visual Studio Code wird im Internet auf der folgenden Webseite beschrieben:

https://code.visualstudio.com/docs/azure/docker

Visual Studio Code hat ein Fenster ‚Extensions: Marketplace'. Das wird durch die Tastenkombination <CTRL + SHIFT + X> aktiviert. Dort kann man in einer Liste nach der gewünschten Docker-Erweiterung suchen.

Das entsprechende Listenelement bietet einen Button **[Install]** an (wenn die Komponente noch nicht installiert ist). Durch Klicken auf diesen Button wird die Installation der gewünschten Erweiterung gestartet.

## 5.3 Visual Studio 2019 mit Docker Development Tools

Eine sehr komfortable IDE ist Visual Studio 2019 (oder höher) mit aktivierten integrierten Docker-Tools. Visual Studio 2019 erlaubt es, eigene Docker Applikationen direkt in der gewählten Docker-Umgebung zu entwickeln, auszuführen und zu überprüfen.

Visual Studio 2019 ist sowohl für Windows- als auch für Mac-Computer verfügbar.



Docker-Anwendungen, die aus einzelnen oder mehreren Containern bestehen, können direkt auf einem Docker Host ausgeführt und gedebugt werden.

Applikationen können bearbeitet und aktualisiert werden, ohne dass ein Container neu erstellt werden muss. Mit dieser IDE können Docker Container sowohl für Linux als auch für Windows erstellt und bearbeitet werden.

Auch die Unterstützung von Kubernetes ist jetzt in der Microsoft Azure-Workload enthalten.

Zusätzlich gibt es natürlich den bekannten Komfort bei der Arbeit mit Visual Studio wie IntelliSense, lokale Entwicklung mit vielen gebräuchlichen Emulatoren, GIT Management und Repository Erstellung in der IDE.

Visual Studio 2019 wird auf der Microsoft Homepage für Visual Studio in drei verschiedenen Versionen zum Download angeboten:

| | |
|---|---|
| Community 2019 | Eine kostenlose Version für Einzelentwickler, für akademische Nutzung und für die Entwicklung von Open-Source-Anwendungen. |
| Professional 2019 | Eine kostenlose Testversion für die Einzelnutzung. |
| Enterprise 2019 | Eine kostenlose Testversion für Organisationen. |

Alle drei Varianten können für Windows über die Webseite mit folgender URI heruntergeladen werden:

https://visualstudio.microsoft.com/de/vs/

Über die folgende URI kann Visual Studio 2019 für Mac installiert werden:

https://docs.microsoft.com/de-de/visualstudio/mac/?view=vsmac-2019

Bei der Installation ist es möglich, die Unterstützung für zahlreiche Technologien auszuwählen, darunter C++, Node.js, Python, .NET, JavaScript, TypeScript usw.

### 5.3.1 Installation von Visual Studio für die Arbeit mit Docker

Bei der Installation von Visual Studio sind die benötigten Erweiterungen vor dem eigentlichen Start des Setups auszuwählen.

Besonders ist darauf zu achten, dass bei der Installation mit dem Visual Studio Installer auf der Seite EINZELNE KOMPONENTEN die Option *Containerentwicklungstools* aktiviert ist, wie das im folgenden Screenshot zu sehen ist.

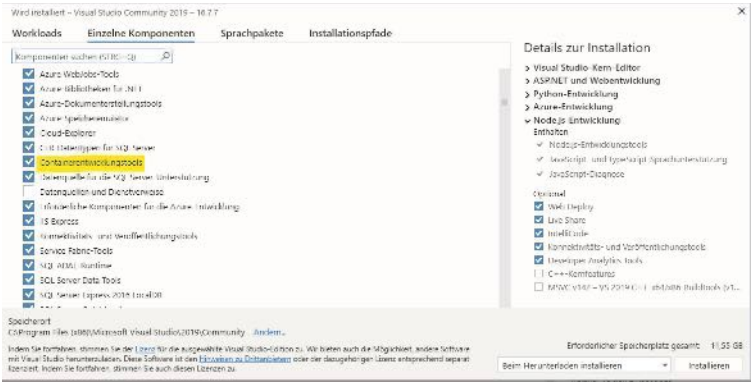



**Abb. 5.1** Visual Studio Setup-Dialog

Eine weitere Voraussetzung für die Entwicklung von Docker ist auch hier, dass „Docker Desktop für Windows“ installiert ist.

## 5.4 Eclipse und Docker

Bei Doclipser handelt es sich um ein Plug-in für Eclipse (Docker + Eclipse = Doclipser), welches bei Zenika entwickelt wurde. Dieses Plugin stellt Docker-Support in der Eclipse IDE zur Verfügung. Docker wird hierbei direkt in der Eclipse IDE integriert. Doclipser erlaubt es auch, Docker Container direkt aus der Eclipse IDE heraus zu starten.



### 5.4.1 Installation von Doclipser

Als Voraussetzung benötigt man eine Eclipse Installation auf dem Entwicklungssystem. Doclipser wird dann dort als Plugin installiert. Es muss der gewöhnliche Installationsvorgang über die Updateseite durchgeführt werden.

Mit dem Eclipse Update Manager geht das folgendermaßen:

- Gehen Sie auf HELP
- Wählen Sie anschließend INSTALL NEW SOFTWARE
  Es erscheint der Dialog „INSTALL“.
- Klicken Sie dort auf den Button [ADD ...]
  Es erscheint der Dialog „ADD REPOSITORY“
- Geben Sie hier im Feld *Name* „Doclipser“ und im Feld *Location* die folgende URI ein:
  https://dl.bintray.com/zenika/doclipser/
  und bestätigen Sie danach mit dem Button [OK].
- Warten Sie im Installationsdialog ab, bis das Listenfeld „Name“ aktualisiert ist.
- Wählen Sie die Plug-Ins aus und klicken sie den Button [NEXT].

- Akzeptieren Sie zuletzt die Lizenzbedingungen und schließen Sie die Installation mit dem Button [FINISH] ab.

Danach sind die beiden neuen Doclipser Features verfügbar:

- Der Dockerfile Editor
- Der Docker API Client

### 5.4.2 Editieren von Dockerfiles

Der Dockerfile Editor von Doclipser bietet Funktionen wie Syntax Highlighting, Autovervollständigung und Syntaxüberprüfung bei der Erstellung und Bearbeitung von Anweisungen in Dockerfiles.

### 5.4.3 Steuerung von Containern

Doclipser bietet darüber hinaus die Funktionalität aus der Eclipse GUI heraus Docker Container zu kontrollieren.

Es werden die Menü-Kommandos angeboten:

- Docker Build – Erstellt ein Docker Image. Es muss dabei ein Verzeichnis angegeben werden, welches das Dockerfile enthält.
- Docker Run – Ausführung der Docker Build Anweisung mit anschließender Ausführung des Images.
- Docker Logs – Zeigt die Logs eines Docker Containers an.
- Docker PS – Listet die aktuell laufenden Container.
- Docker rm – Entfernt einen Docker Container

## 5.5 Curl

Curl steht als Abkürzung für den Namen „Client for URLs“. Es handelt sich hier um ein Kommandozeilen-Tool zur Übertragung von Informationen über eine Internetadresse. Dabei können auch POST-Übertra-

gungen durchgeführt werden. Es werden mittlerweile zahlreiche Protokolle unterstützt, wie zum Beispiel HTTP, HTTPS, FTP, FTPS, DICT, LDAP, RTMP und Gopher...

Bei LINUX ist curl schon lange Bestandteil der meisten LINUX-Distributionen.

Im April 2018 wurde curl nun auch als integrierte Anwendung bei Windows 10 aufgenommen und wird standardmäßig zusammen mit dem Betriebssystem installiert.



Es handelt sich hier zunächst nicht um ein spezielles Docker Tool. Curl ist aber sehr hilfreich bei der Entwicklung von Microservices, die in Docker Containern laufen. Diese besitzen oft keine HTML-Schnittstelle. Somit ist keine direkte Kommunikation über einen Web-Browser mit so einem Service möglich. Das Tool curl erlaubt es uns aber, eine URI im http-Format anzusprechen und die Antwort eines Services auf diesen Aufruf anzuzeigen.

Hier ein kurzes Beispiel für die Kommunikation mit einem Service, der in einem Container läuft.

Wir starten dazu einen unserer NGINX Container, der das Beispiel Image ‚hello-web' ausführt.

Hier noch einmal das Kommando, um den Container zu starten:

```
1  > docker run --name hello_web -d -p 8880:80 '
2  <DOCKER_ID>/hello-web
```

Falls noch ein Container mit diesem Namen existiert, dann antwortet Docker mit einer Fehlermeldung. In diesem Fall müssen wir den Container vor der Ausführung (wie schon gehabt) wieder löschen.

Beispiel:

```
1  > docker container rm hello_web
```

Danach sollte das oben angegebene Kommando dann auch ohne Probleme ausgeführt werden.

Wenn wir im Web-Browser in der Adresszeile jetzt wieder die passende URI

http://localhost:8880/

eingeben, wird dort auch unsere Webseite angezeigt (siehe Abb. 4.27 Die Webseite des NGINX Containersrs.).

Wir können aber auch diese URI als Parameter an curl übergeben und sehen dann die Informationen, die bei der Kommunikation mit unserem Web-Service ausgetauscht werden.

Starten Sie also eine Kommando-Shell und geben Sie das folgende Kommando ein:

```
1  > curl  http://localhost:8880/
```

Curl liefert Ihnen dann als Ergebnis die folgende Ausgabe:

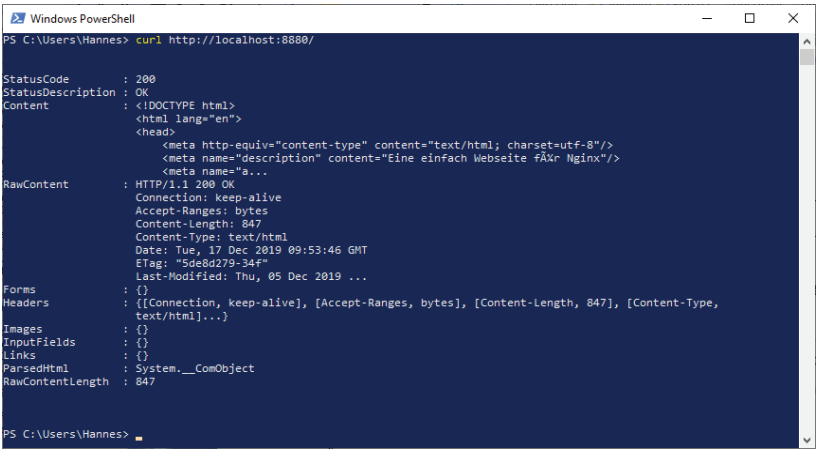



**Abb. 5.2** Shell Ausgabe bei curl-Test

Hier können wir sehen, dass unser Service mit dem Status 200 geantwortet hat, was OK bedeutet. Der Dokument-Typ ist html, als Sprache ist Englisch eingestellt. Auch die Meta-Informationen werden hier teilweise angezeigt.

Vergleichen Sie doch einfach einmal die ausgegebenen Informationen mit der Datei ‚index.html' aus unserem „Hello-Web"-Beispiel.

### 5.5.1 Curl-Hilfe



Für curl gibt es mittlerweile sehr viele Kommandozeilen-Parameter. Da curl als Open-Source-Projekt immer weiter entwickelt wird, kann man davon ausgehen, dass die Anzahl der verfügbaren Parameter in zukünftigen Versionen immer mehr werden.

Um eine Übersicht über die aktuell verfügbaren Kommandozeilen-Parameter und deren Funktion zu bekommen, gibt es zwei Möglichkeiten.

Eine stellt den Aufruf mit dem Parameter --help bzw. die Kurzform –h dar.

Die andere Möglichkeit ist die Anzeige des Manuals (der gesamten man Pages). Dies geht durch Aufruf mit dem Parameter `curl --manual`.

### 5.5.2 Die wichtigsten curl-Parameter

Die Grundform eines `curl`-Kommandos sieht so aus:

```
1  > curl [optionen] [<URI> …]
```

Die einfachste Variante ist dabei wohl

```
1  > curl [URI]
```

Zum Beispiel:

```
1  > curl http://example.com
```

Sie können das Beispiel ruhig einmal ausprobieren. Es gibt diese Domäne wirklich. Sie wurde ausschließlich für Demonstrationszwecke angelegt und darf in Dokumentationen ohne besondere Erlaubnis angegeben werden.

Der obige Aufruf zeigt so nicht alle Informationen. Wenn wir mehr sehen wollen, müssen wir den Verbose Mode aktivieren. Im diesem Modus ist `curl` wesentlich gesprächiger.

Ausgabe des Ergebnisses im Verbose-Modus:

```
1 > curl -v http://example.com
```

oder

```
1 > curl --verbose http://example.com
```

Wenn das curl-Kommando recht umfangreiche Daten ausgibt, möchte man diese vielleicht in einer Datei sammeln. Das wird durch den Parameter --o oder --output bewirkt.

Das folgende Kommando speichert die Ausgabe in der Datei ‚log.txt':

```
1 > curl -v http://example.com -o log.txt
```

Wir wollen natürlich auch Daten an eine Web-Applikation senden. Das http-Protokoll hat dafür die Methode POST vorgesehen. Bei curl gibt es dafür den Kommandozeilen-Parameter `-d` beziehungsweise --data. Damit können Name/Wert-Paare an einen Web-Service übertragen werden.

Hier ein Beispiel (das Beispiel funktioniert im Moment in der Realität aber leider nicht):

```
1 > curl -d surname=Hannes http://www.telephonelist.com
```

Kapitel 6

# Docker-Architektur

Sie haben mittlerweile recht viele Docker-Begriffe kennengelernt. Da gibt es Images, Container, eine Engine, den Host, Registries oder den Docker Hub. Möglicherweise sind Sie jetzt etwas verwirrt von den vielen Begriffen.

Das folgende Diagramm soll Ihnen daher eine Übersicht über das Gelernte geben und die Zusammenhänge zwischen den verschiedenen Docker-Elementen verdeutlichen:

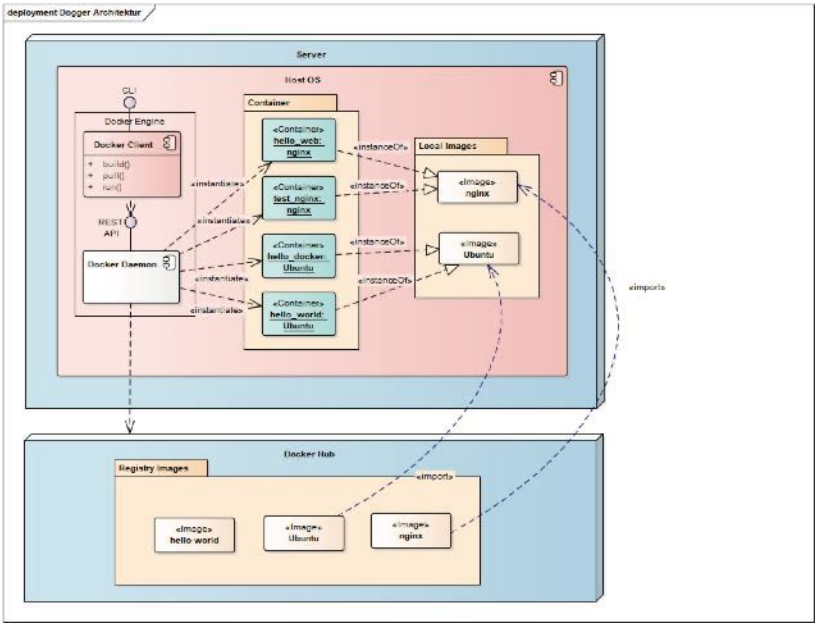



**Abb. 6.1** Übersicht über die Docker-Architektur

Das Diagramm zeigt zwei große Blöcke. Da ist ein Block, der den Server repräsentiert, und ein anderer steht für den Docker Hub.

Das aktive Server-Element ist die Docker Engine. Wird ein Container gestartet, so sucht die Docker Engine lokal nach dem zugehörigen Image. Ist dies lokal nicht vorhanden, so sucht die Docker Engine in einer Registry, standardmäßig ist das der Docker Hub, nach dem Image. Die Docker Engine erstellt dann lokal auf dem Server eine Kopie, instanziiert und startet für dieses Image einen Container.

## 6.1 Die Docker Engine



Die Docker Engine wurde in einer klassischen Client-Server-Architektur entworfen. Da kommuniziert ein Docker Client mit einem Docker Daemon. Als Kommunikations-Schnittstelle dient ein REST-API.

Die Abkürzung REST steht für Representational State Transfer. Ein API ist eine Programmier-Schnittstelle (Application Programming Interface). Ein REST API definiert eine Programmierschnittstelle, die beschreibt, wie verteilte Systeme, also Anwendungen, deren Komponeneten auf verschiedenen Computern liegen, miteinander kommunizieren können.

Es ist daher nicht zwingend notwendig, dass Docker Client und Docker Daemon auf dem gleichen System ausgeführt werden. Der Docker Client kann durchaus auf einem anderen System laufen und sich mit dem Docker Daemon remote verbinden.

Dem Anwender wird zur Kommunikation mit dem Docker Client eine Kommando-Schnittstelle (CLI – Command Line Interface) zur Verfügung gestellt. Darüber haben wir ja schon Kommandos wie `docker run`, `docker build` oder `docker pull` eingegeben.

Der Docker Client reicht ein über die CLI Schnittstelle eingegebenes Kommando über das REST API an den Docker Daemon weiter. Dieser übernimmt dann die eigentlichen Aufgaben wie den Bau von Contai-

nern aus den Images, die Ausführung von Containern oder die Verteilung von Containern. Der Docker Daemon kann auch mit anderen Daemons kommunizieren, um Docker-Dienste zu managen.

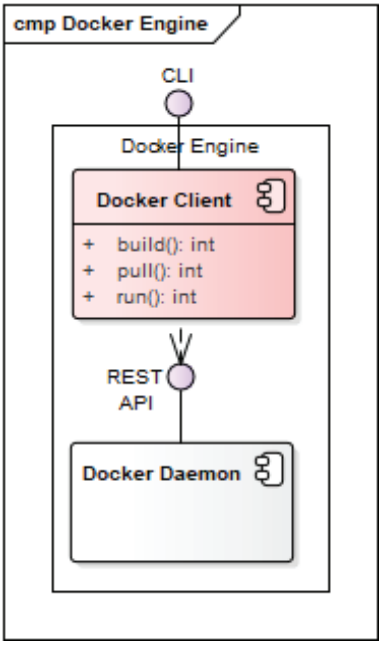



**Abb. 6.2** Die Docker Engine

## 6.2 Docker Images und Registries

Wie bereits zu Beginn erwähnt, speichert und verwaltet eine Docker Registry die Docker Images.

Docker Hub ist eine öffentliche Registry, die jeder nutzen kann. Es können aber auch andere Registries verwendet werden. Docker Hub ist im Normalfall als Standard Registry konfiguriert.

Es besteht aber durchaus auch die Möglichkeit, eigene, private Registries anzulegen und zu benutzen.

Die Docker CLI Kommandos `docker pull` und `docker run` holen bei ihrer Ausführung die benötigten Images automatisch von der aktuell konfigurierten Registry.

Führt man das Kommando `docker push` aus, legt der Docker Daemon das angegebene Image in der konfigurierten Registry ab.

## 6.3 Docker Container

Wie bereits erwähnt, handelt es sich bei einem Docker Container um die lauffähige Instanz eines Docker Images. Diese wird durch den Docker Daemon instanziiert und ausgeführt. Der Docker Daemon wiederum ist Bestandteil der Docker Engine, die auf dem Host System läuft.



Der Container beinhaltet zuerst das Ziel-Betriebssystem. Zusätzlich sind alle Binärdateien und Bibliotheken, welche zur Ausführung einer Applikation benötigt werden, enthalten. Nicht zuletzt sind die Applikation selbst Bestandteil eines Containers.

Das nächste Diagramm zeigt Ihnen die interne Architektur von Docker Containern.

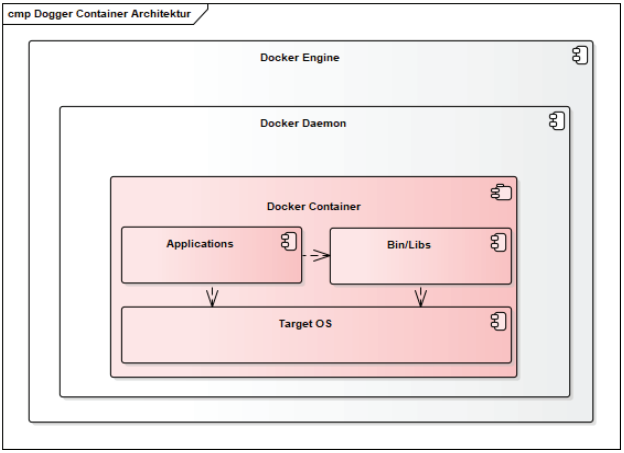



**Abb. 6.3** Die Docker Container-Architektur

Kapitel 7

# Bewährte Praktiken bei der Arbeit mit Docker

## 7.1 Schreiben von Dockerfiles

### 7.1.1 Die Reihenfolge im Dockerfile

Achten Sie beim Erstellen eines Dockerfiles auf eine sinnvolle Reihenfolge der Docker-Anweisungen. Docker arbeitet das Dockerfile von oben nach unten ab und jede Anweisung erstellt eine neues Layer, was wiederum einem neuen Image entspricht, welches als Basis für die nächste Anweisung im Dockerfile herangezogen wird. Die temporären Images werden in einem Cache-Speicher verwaltet, um den Zugriff darauf zu beschleunigen.



Versucht eine Docker-Anweisung im Image auf ein Verzeichnis zuzugreifen, welches erst in einer späteren Anweisung erstellt wird, so schlägt der Bau eines Images fehl.

Bei der Reihenfolge sollte man auch beachten, dass Docker-Anweisungen, die den Docker Cache seltener verändern, im Dockerfile weiter oben stehen als jene, die häufiger Veränderungen verursachen. Die Reihenfolge im Dockerfile wird also durch die Häufigkeit der verursachten Veränderung im Cache bestimmt.

### 7.1.2 Gruppierung verwandter Build-Anweisungen

Man kann zum Beispiel beim RUN-Befehl genau wie bei Shell-Kommandos auch mehrere Befehle mit &&-Zeichen verknüpfen. Dadurch wird die Anzahl der temporären Images im Cache deutlich reduziert.

Man kann die folgenden beiden RUN-Anweisungen zwar nacheinander angeben:

```
1 RUN apt get update
2 RUN apt get install openjdk-8-jdk ssh vim
```

Besser ist es aber die beiden Kommandos zusammen, mit einer einzigen RUN-Anweisung auszuführen:

```
1 RUN apt get update && apt get install openjdk-8-jdk ssh vim
```

### 7.1.3 Halten Sie Ihre Images klein

Kopieren Sie keine unnötigen Daten in das Image.

Vermeiden Sie die Installation von unnötigen Applikationen. Installieren Sie z.B. keine Debugging Tools, wenn diese nicht unbedingt benötigt werden.

ACHTUNG! Bestimmte Paket-Manager, wie zum Beispiel `apt`, installieren automatisch Pakete mit, die eine Abhängigkeit zum gerade installierten Paket besitzen und als „empfohlene Pakete“ markiert sind. Das kann bei `apt` durch die Angabe des Flags `–no-install-recommends` als Parameter verhindert werden.

Beispiel:

```
1 apt-get --no-install-recommends install python3.7
```

Weiterhin verwalten die meisten Paket-Manager ihren eigenen Cache-Speicher, der unter Umständen im Image bleibt, obwohl er hier nach der Installation völlig unnötig ist. Man kann diesen Cache wieder löschen, allerdings nur in der gleichen RUN-Anweisung, die ihn erzeugt hat. Wird versucht, den Cache in einer nachfolgenden RUN-Anweisung zu entfernen, so verringert dies die Größe des Images nicht.

Beispiel:

```
RUN apt get update \
 && apt get install --no-install-recommends \
openjdk-8-jdk \
 && rm -rf /var/lib/apt/lists/*
```

### 7.1.4 Verbessern Sie die Wartbarkeit Ihrer Images

Verwenden Sie die offiziellen Images, wann immer dies möglich ist. In offiziellen Images sind alle nötigen Installationsschritte bereits enthalten und bewährte Praktiken werden angewandt.

Geben Sie im Dockerfile nicht „latest“ als Tag für Ihre Version an, sondern bestimmen sie die Version genau.



Beispiel:

```
FROM ubuntu:18.4
```

Nutzen sie aber, wann immer möglich, die aktuellste Version eines offiziellen Docker Images als Basis für Ihr eigenes Image.

## 7.2 Entkoppeln Sie die Komponenten

Jeder Container sollte nur für die Erfüllung einer einzigen Aufgabe entwickelt werden. Werden die verschiedenen Funktionen einer Anwendung aufgebrochen und als unabhängige Tasks auf mehrere Container verteilt, so erleichtert das die Skalierung der Services und erhöht die Wiederverwendbarkeit der entwickelten Komponenten.

So könnte zum Beispiel eine Web-Applikation aus drei verschiedenen Containern zusammengesetzt sein, die alle Ihre eigenen, unabhängigen Images besitzen. Dabei könnte sich ein Image um das Web Interface kümmern, das andere könnte den Zugriff auf die Datenbank organisieren und ein drittes Image ist für die Organisation eines In-Memory Cache Speichers zuständig.

Wenn in jedem Container nur ein Task ausgeführt würde, dann wäre das ein idealer Entwurf. Aber es gibt natürlich manchmal auch Gründe, dass in einem Container mehrere Tasks laufen müssen.

## 7.3 Vergeben Sie Tags für Ihre Images

Wie Sie im Docker Hub sehen können, werden Docker Images üblicherweise durch zwei Angaben identifiziert. Die eine Angabe ist der Image-Name, die andere ist der Tag. Die Angaben von Name und Tag werden durch einen Doppelpunkt getrennt dargestellt. Über den Tag identifiziert man üblicherweise die Version eines Images.

Als Form für Versionsnummern wird im Allgemeinen die sogenannte „Semantische Versionierung" empfohlen. Bei dieser Form besteht eine Versionsnummer aus drei Teilen, die durch einen Punkt getrennt sind. Diese Form hat den folgenden Aufbau:

<Hauptversion>.<Nebenversion>.<Patchversion>

Beispiel für die Angabe eines Docker Images mit Versionsinformation:

```
1 nginx:1.17.7
```

- Der Wert der <Hauptversion> wird nur bei Änderungen erhöht, bei denen sich das Interface so verändert, dass es zur Vorgänger-Version nicht mehr kompatibel ist.
- Der Wert der <Nebenversion> wird erhöht, wenn neue Features implementiert wurden.
- Der Wert der <Patchversion> ändert sich, wenn Bugs behoben wurden.

Beispiel - Bauen des „hello-web" Images mit Tag 1.0.0:

```
1 > docker build -t <DOCKER_ID>/hello-web:1.0.0 .
```

Beispiel – Veröffentlichen des „hello-web“ Images in Docker Hub mit Tag 1.0.0:

```
1  > docker push <DOCKER_ID>/hello-web:1.0.0
```

## 7.4 Verwenden Sie COPY anstelle von ADD

Für das Dockerfile stehen zwei Kommandos zum Kopieren von Dateien von einem Host in ein Docker Image zur Verfügung. Zum einen das Kommando `COPY`, zum anderen das Kommando `ADD`.

COPY – Dieses Kommando kopiert rekursiv lokale Dateien oder Verzeichnisse aus dem Host in das Image.

ADD – Das ADD Kommando bietet die gleiche Funktion wie COPY mit der Erweiterung, dass man als Datenquelle auch URIs angeben kann.



Wenn bei ADD entfernte URIs als Datenquelle angegeben werden, so eröffnet dies die Möglichkeit, dass in den Datentransfer eingegriffen werden kann. Dadurch hätten Hacker die Möglichkeit, die zu übertragenen Daten zu manipulieren.

Sicherer ist es, die Daten vorher über eine sichere TLS-Verbindung zu laden, auf Sicherheitsrisiken zu prüfen und dann im „Build Context“ abzulegen. Das Übertragen ins Image kann dann ohne Risiko über ein COPY-Kommando erfolgen.

Kapitel 8

# Daten speichern in Docker

Alle Daten, die bei Ausführung im Dateisystem eines Containers entstehen, werden innerhalb des Containers gespeichert. Der Zugriff auf diese Daten im Container von außen ist dadurch sehr schwierig (z.B. der Zugriff auf Log-Dateien). Daten aus einem Container können wiederum nicht ohne Weiteres an eine andere Stelle verschoben werden, da diese eng mit dem Container verbunden sind. Das gleiche Problem gilt für das Verändern der Daten in einem Container von außerhalb.



Das bedeutet aber auch, dass diese Daten verloren gehen, wenn ein Container nicht mehr länger existiert. Wenn Sie einen Container mit dem Befehl `docker container rm <container_name>` entfernen, dann löschen Sie dadurch auch alle Daten, die während der Ausführung des Containers entstanden sind. Befinden sich in einer Datenbank neue Einträge, dann verschwinden diese für immer zusammen mit Ihrem Container.

Auch wenn der Container anschließend aus dem zugrundeliegenden Image neu gebaut werden kann, sind dort nur die Daten vorhanden, die über die Anweisungen im Image erstellt wurden.

Um dieses Problem zu umgehen, bietet Docker zwei Möglichkeiten, Daten in einem Container persistent, also nicht flüchtig, zu speichern. Eine Möglichkeit ist die Verwendung von Docker Volumes, die andere nennt sich „Bind Mount".

Volumes kann man sich am ehesten wie einen externen Datenträger vorstellen. Etwa so wie ein USB-Stick, den man an verschiedene Computer anschließen kann, um auf die darauf gespeicherten Daten zuzugreifen.

Beim „Bind Mount“ wird ein Teil des Host-Dateisystems, das kann ein ganzes Verzeichnis oder eine einzelne Datei sein, in das Dateisystem des Docker Containers eingebunden. Das kann man mit der Mount-Funktionalität vergleichen, die der Linux-Befehl `mount` bereitstellt.

## 8.1 Docker Volumes

Docker Volumes sind in der Regel den Bind Mounts vorzuziehen, da diese sowohl auf Linux Containern als auch auf Windows Containern verwendet werden können. Damit können Container dieser beiden Typen persistente Daten gemeinsam nutzen. Zudem können Volumes allgemein leichter auf mehrere Container verteilt werden und sind darüber hinaus auch einfacher zu sichern.

### 8.1.1 Docker Volume erzeugen

Jetzt kommen wir zur praktischen Arbeit mit Volumes. Als Erstes erstellen wir mit Docker ein neues Volume. Dafür gibt es ein eigenes Docker CLI-Kommando. Hier die Syntax für dieses Kommando:

```
1 docker volume create <VOLUME_NAME>
```

Für den Namen eines Volumes sollten folgende Regeln beachtet werden:

Das erste Zeichen des Namens muss alphanumerisch sein [a-zA-Z0-9]. Der Rest der Zeichenkette muss auch aus alphanumerischen bestehen und kann auch zusätzlich die Sonderzeichen Unterstrich, Punkt und Bindestrich enthalten [a-zA-Z0-9_.-].

Wir erstellen mit diesem Kommando ein neues Docker Volume mit dem Namen „test-vol“.

Dazu starten wir eine Shell und geben folgendes Kommando ein:

```
1 docker volume create test-vol
```

Um uns alle vorhandenen Docker Volumes anzeigen zu lassen, geben wir dieses Kommando ein:

```
1  docker volumes ls
```

Damit wir von einem Volume detailliertere Informationen bekommen, gibt es das CLI-Kommando `docker volume inspect`. Hier die Syntax:

```
1  docker volume inspect <VOLUME_NAME>
```

Um die Informationen über das von uns neu erstellte Volume zu erhalten, geben wir den folgenden Befehl in die Kommandozeile ein:

```
1  docker volume inspect test-vol
```



Der folgende Screenshot zeigt, wie die Ausgaben der oben beschriebenen Aktionen im PowerShell-Fenster aussehen (Abb. 8.1).

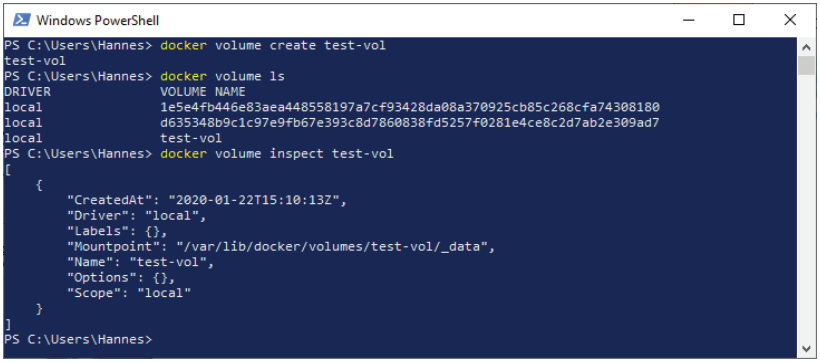



**Abb. 8.1** Befehle zur Arbeit mit Docker Volumes

### 8.1.2 Docker Volume in Container einbinden

Wenn man ein Docker Volume in einen Docker Container einbinden will, dann muss man den Volume-Namen und das Zielverzeichnis im Container beim Start des Containers mit angeben.

Das Kommando `docker run` erlaubt heutzutage beim Aufruf die Verwendung von zwei unterschiedlichen Flags, um diese Funktionalität zu erreichen.

```
1 -v oder --volume
```

Die Syntax für dieses Flag hat diese Form:

```
1 -v | --volume [<VOLUME_NAME >]:<mount_path>[:<options>]
```

Diese Form besteht nach dem Flag aus drei Feldern, die durch einen Doppelpunkt getrennt werden. Diese Felder müssen in der vorbestimmten Reihenfolge eingegeben werden. Für Einsteiger ist die Bedeutung dieser Felder nicht unbedingt offensichtlich.

- Das erste Feld enthält den Namen des Volumes, das eingebunden werden soll. Wenn ein anonymes Volume im Container erstellt werden soll, kann diese Angabe weggelassen werden.
- Das zweite Feld enthält eine Pfadangabe im Container. Dort wird das Volume eingebunden.
- Das dritte Feld ist optional. Es können hier zusätzliche Optionen, durch Komma getrennt, eingegeben werden (z.B. ro für read only).

```
1 --mount
```

Die Syntax für das `mount` Flag sieht folgendermaßen aus:

```
1 --mount <key>=<value> [,<key>=<value> [,…]]
```

Das `mount` Flag wird mit mehreren Key-Value-Paaren als Parameter angegeben. Diese Key-Value-Paare sind durch Komma getrennt und können in beliebiger Reihenfolge auftreten. Dadurch ist diese Form leichter zu verstehen und wird für Docker-Einsteiger gerne empfohlen.

Die folgende Liste zeigt die verfügbaren Keys und die Werte, die dazu angegeben werden können:

`type` - bestimmt den Mount Typ. Ihm kann eine der folgenden Angaben zugewiesen werden: `bind`, `volume`, `tmpfs`. Bei Volumes steht hier dann `volume`.

`source` - hier wird der Name des Volumes zugewiesen.

`destination` - weisen Sie hier den Pfad zum Zielverzeichnis im Container-Dateisystem zu.

`readonly` das ist eine Option, die keine Wertzuweisung braucht. Mit dieser Angabe wird das Volume im Container mit Schreibschutz angelegt.



Jetzt kommt ein praktisches Beispiel für ein `docker run`-Kommando, das einen Container auf Basis des Ubuntu Images aus dem Docker Hub erstellt (siehe Kapitel 4.5.1). Der Container erhält den Namen „voltest“ und es wird unser oben erstelltes Volume „test-vol“ ins Container-Verzeichnis `/test_data` eingebunden. Bei diesem Beispiel kommt die Variante mit dem `--mount` Flag zur Anwendung.

Wir geben mit dem letzten Parameter wieder an, dass beim Container-Start mit `/bin/bash` eine „Bash Shell“ gestartet werden soll, über die wir dann anschließend Änderungen im Container vornehmen können.

Folgendes Kommando wird in das Kommando-Fenster eingegeben:

```
> docker run -i -t \
--name=voltest \
--mount source=test-vol,target=/test_data \
ubuntu /bin/bash
```

Hinweis: Das obige Kommando wurde der Übersichtlichkeit halber auf mehrere Zeilen verteilt. Um dies zu verdeutlichen, steht bei diesem Beispiel am Ende der Zeile ein Backslash \ (wie man das von Linux-Kommandofenstern kennt). Sie können das Kommando aber in einer einzigen Zeile eingeben, ohne den Backslash. Falls Sie in der PowerShell mehrzeilige Eingaben vornehmen wollen, so verwenden Sie am Ende

einer Zeile das Backtick-Zeichen (`), um den Zeilenumbruch zu maskieren.

Wenn der Ubuntu System Prompt # erscheint, geben wir das Linux-Kommando `ls` ein, um die Verzeichnisse im Container anzuzeigen. Wir können sehen, dass es jetzt auch ein Verzeichnis `test_data` gibt.

Wir wechseln in dieses Verzeichnis (`cd test_data`) und sehen uns an, was sich dort befindet (im Moment noch nichts).

Wir erzeugen jetzt im Verzeichnis test_data eine neue Datei. Geben Sie dazu das folgende Kommando ein (das # Zeichen am Anfang soll den Prompt darstellen und wird nicht mit eingegeben):

```
1  # echo „Hello World" > hello.txt
```

Damit wird eine Datei mit dem Namen `hello.txt` erzeugt. Diese enthält den Text „Hello World".

Mit dem Kommando

```
1  # cat hello.txt
```

können Sie das Ergebnis mit dem `ls`-Kommando überprüfen.

Verlassen sie die `bash shell` mit dem Kommando `exit`.

Der folgende Screenshot zeigt die Ausführung der obigen Kommandos in einem PowerShell-Fenster (Abb. 8.2).

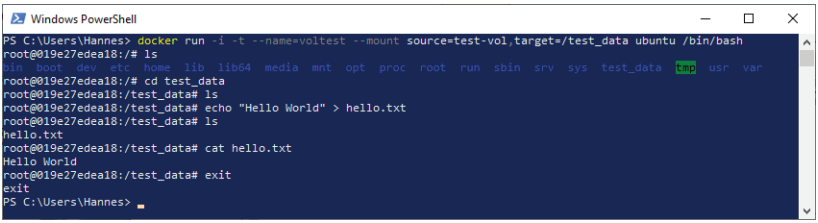



**Abb. 8.2** Docker Volume in Container mit Parameter **`–mount`** einbinden

Das nächste Beispiel für ein `docker run`-Kommando ist ähnlich wie die erste Variante. Es wird ein neuer unabhängiger Container mit dem Namen „voltest2“ erstellt. Auch hier wird wieder unser oben erstelltes Volume „test-vol“ ins Container-Verzeichnis `/test_data` eingebunden. Bei diesem Beispiel kommt die Variante mit dem `-v` Flag zum Einsatz.

```
> docker run -i -t '
--name=voltest2 `
-v test-vol:/test_data '
ubuntu /bin/bash
```

Wenn wir den neuen Container erneut untersuchen, dann stellen wir fest, dass hier ebenfalls ein Verzeichnis `test_data` vorhanden ist und dort ebenfalls die Datei `hello.txt` mit gleichem Inhalt existiert.



Unsere zwei Container-Instanzen haben jetzt offensichtlich Zugriff auf gemeinsame Daten.

Hier noch ein Screenshot mit dem zweiten Beispiel.

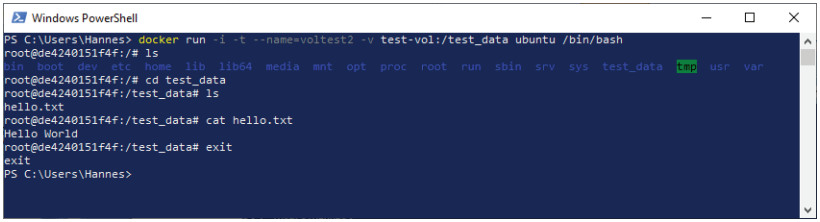



**Abb. 8.3** Docker Volume in Container mit Parameter –v einbinden

### 8.1.3 Docker Volume entfernen

Zuletzt wollen wir unser Docker Volume wieder entfernen. Das Kommando zum Entfernen eines Containers hat folgende Form:

```
> docker volume rm <volume_name>
```

Wenn das Volume noch in irgendeinem Container benutzt wird, schlägt das Löschen fehl. Probieren wir das jetzt einmal aus:

```
1  > docker volume rm test-vol
```

Erst müssen alle Container, welche das Volume benutzen, gelöscht werden. Danach ist Löschen eines Volumes möglich.

```
1  > docker container rm voltest
2  > docker container rm voltest2
```

Nun wird das Kommando

```
1  > docker volume rm test-vol
```

ohne Fehlermeldung ausgeführt. Das Volume ist danach nicht mehr vorhanden und die Daten können damit auch nicht wiederhergestellt werden. Als Bestätigung wird nach dem erfolgreichen Entfernen eines Volumes dessen Name ausgegeben.

Mit dem Kommando

```
1  > docker volume ls
```

können Sie überprüfen, ob das Volume auch wirklich gelöscht wurde (Abb. 8.4).

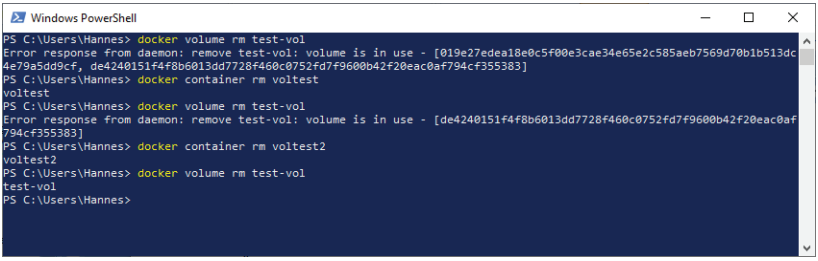



**Abb. 8.4** Docker Volume entfernen

## 8.2 Bind Mounts

„Bind Mounts“ waren zu Beginn der Docker-Ära die erste Möglichkeit, mit persistenten Daten umzugehen. Sie haben aber gegenüber Docker Volumes einige Nachteile und eingeschränkte Funktionalitäten.

Beim „Bind Mount“ wird entweder eine Datei oder ein Verzeichnis aus dem Dateisystem des Host Computers in das Dateisystem des Containers montiert. Dieses Verzeichnis bzw. diese Datei wird im Container durch die komplette Pfadangabe referenziert. Es kann entweder der absolute Pfad oder der relative Pfad der Quelldaten auf dem Host-Rechner angegeben werden.

Im Gegensatz dazu befindet sich ein Volume als Verzeichnis innerhalb eines speziellen Docker-Speichers auf dem Host-Rechner, dessen Inhalte von Docker selbst verwaltet werden.



Falls im Übrigen ein Verzeichnis auf dem Host-System noch nicht existiert, wird es von Docker bei Bedarf automatisch neu angelegt.

### 8.2.1 Windows Host-Computer für „Bind Mount“ vorbereiten

Damit Sie „Bind Mount“ auf Ihrem Windows Host-Rechner verwenden können, müssen Sie diesen zuerst entsprechend konfigurieren. Unter Windows klicken Sie dazu auf das Docker-Symbol in der Task-Leiste, um ein Kontext-Menü zu öffnen. Hier wählen Sie den Menüpunkt Settings (Abb. 8.5).

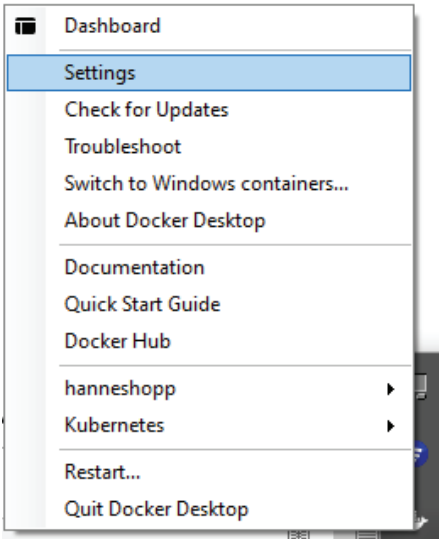



**Abb. 8.5** Menüpunkt SETTINGS im Docker Kontext-Menü.

Im Dialogfenster „SETTINGS“ wählen Sie das Register [SHARED DRIVES] (Abb. 8.6).

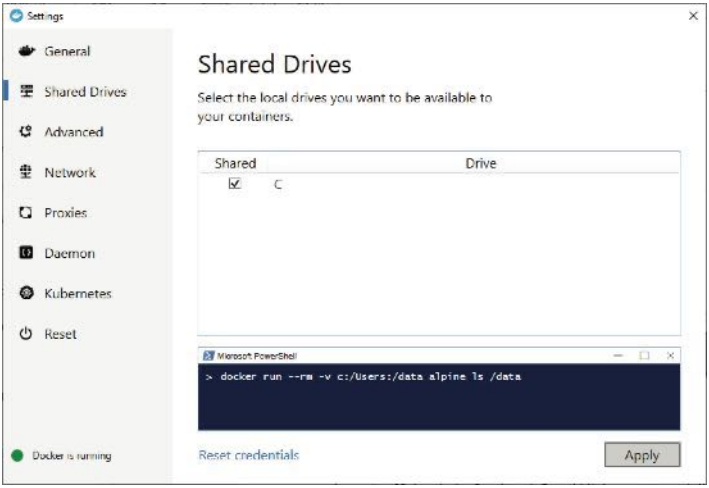



**Abb. 8.6** Register „SHARED DRIVES“ im Docker „SETTINGS“-Dialog

Hier aktivieren Sie das Kontrollkästchen *Shared* für das gewünschte Laufwerk, das die Daten für den „Bind Mount" enthält. Zuletzt bestätigen Sie die Einstellung durch einen Mausklick auf den Button [Apply].

### 8.2.2 „Bind Mount" beim Start eines Containers angeben

Wenn man eine Datei oder ein Verzeichnis aus dem Host-System in einen Docker Container einbinden will, dann muss man beim Start des Containers folgende Informationen angeben:

- Den Pfad zur gewünschten Datei oder zu dem Verzeichnis auf dem Dateisystem des Host-Rechners.
- Das Zielverzeichnis im Container, in dem die Datei oder das Verzeichnis montiert werden soll.



Beim Kommando `docker run` gibt es auch hier beim Aufruf die Verwendung der zwei unterschiedlichen Flags.

```
1 -v oder --volume
```

Die Syntax für dieses Flag hat hier diese Form:

```
1 -v | --volume [<source_path>]:<mount_path>[:<options>]
```

Diese Form unterscheidet sich von der Volume-Variante nur durch das erste Feld. Dort wird der Volume-Name durch die Angabe des Quellpfandes auf dem Host-Rechner ersetzt

```
1 --mount
```

Die Syntax für dieses Flag ändert sich hier erst einmal nicht:

```
1 --mount <key>=<value> [,<key>=<value> [,…]]
```

Diese Form unterscheidet sich von der Volume-Variante nur durch die beiden Keys `type` und `source`:

`type` - bestimmt den Mount-Typ. Ihm kann eine der folgenden Angaben zugewiesen werden. `bind`, `volume`, `tmpfs`. Bei „Bind Mount" gibt man `bind` als Wert an.

`source` - hier wird die Angabe des Quellpfandes auf dem Host-Rechner als Wert eingetragen.

`destination` - weisen Sie hier den Pfad zum Zielverzeichnis im Container-Dateisystem zu.

`readonly` - das ist eine Option, die keine Wertzuweisung braucht. Mit dieser Angabe wird das Volume im Container mit Schreibschutz angelegt.

Wenn ein Verzeichnis oder eine Datei im Dateisystem des Containers, das als Zielverzeichnis angegeben wird, bereits existiert, dann werden diese von den Host-Daten lediglich überlagert und nicht überschrieben oder gelöscht.

Im folgenden Beispiel nehmen wir unser PHP-Beispiel „Telefon-PHP" aus dem Kapitel 4.9 und binden das Host-Verzeichnis mit der PHP-Datei index.php über den „Bind Mount"-Mechanismus ein.

Dazu wechseln wir in das Verzeichnis **<user_home>/Telefon-PHP**. Hier führen wir das `docker run`-Kommando mit den Angaben zum Montieren des Verzeichnisses `src` ein. Dort befindet sich die Datei `index.php`.

Hier das Beispiel mit der `-v`-Variante:

```
> docker run --name telefon_app '
-d -p 8808:80 '
-v C:\Users\Hannes\Telefon-PHP\src:/var/www/html/ '
<DOCKER_ID>/telefon-app
```

Und zum Vergleich die `–mount`-Variante:

```
docker run --name telefon_app '
-d -p 8808:80 '
--mount type=bind,.'
source=C:\Users\Hannes\Telefon-PHP\src, '
destination=/var/www/html/ '
<DOCKER_ID>/telefon-app
```

Denken Sie daran, für die Angabe `<DOCKER_ID>` Ihre Docker ID einzusetzen, die Sie bei der Anmeldung an Docker Hub verwenden.

Wir starten einen beliebigen Web-Browser und geben in der Adresszeile wieder den URI für localhost und die Portnummer 8808 an.

```
localhost:8808
```

Die Webseite sieht erst einmal genau so aus wie vorher. Wenn wir aber jetzt Änderungen am Code in der Datei `index.php` vornehmen (indem wir zum Beispiel `<body bgcolor="yellow">` angeben oder das Array `userList` mit neuen Einträgen erweitern), dann können wir diese Änderungen sofort nach dem Aktualisieren der Webseite sehen, ohne dass wir den Container stoppen, das Image neu bauen und anschließen den Container neu starten müssen.


**Abb. 8.7** Beispiel zur Ausgabe mit geänderter Datei index.php

Wenn wir zum Abschluss den Container wieder stoppen und entfernen und dann wieder ohne „Bind Mount“ Angaben starten, dann können wir feststellen, dass unsere alte Datei **index.php** im Container-Dateisystem immer noch unverändert ist. Unsere Webseite ist wieder wie vorher.

Probieren Sie das zur Übung ruhig einmal aus. Hier die Docker-Kommandos:

```
> docker stop telefon_app
> docker container rm telefon_app
> docker run --name telefon_app '
-d -p 8808:80 <DOCKER_ID>/telefon-app
```

Vergessen Sie nicht am Ende der Übung wieder aufzuräumen:

Verwenden Sie `docker stop`, um den Container anzuhalten, und `docker container rm`, um ihn zu löschen.

Kapitel 9

# Log-Dateien

Docker Log-Dateien enthalten automatisch geführte Protokolle, die durch Prozesse erstellt werden, welche in den Containers ablaufen. Es werden hier für alle Ereignisse und Aktionen Einträge erstellt, die bei späteren Untersuchungen nützliche Informationen darstellen können. Damit kann bei Problemen die Fehlersuche unterstützt werden, es können aber auch Rückschlüsse über Performanceprobleme aus den Log-Informationen abgeleitet werden.

## 9.1 Container Logs anzeigen

Für die Arbeit mit den Log-Dateien bietet Docker ein Kommando an, mit dem man diese Informationen in einer Kommando-Shell anzeigen kann.

Die Syntax für das CLI-Kommando zur Ausgabe der Container Log-Informationen sieht folgendermaßen aus:

```
1 > docker logs [OPTIONEN] <CONTAINER_NAME>
```

Als Option kann man die folgenden Flags einsetzen:

`--details`
Anzeige weiterer Details, die für ein Log zur Verfügung gestellt werden.

`--follow` oder `-f`
Dieser Parameter ermöglicht die kontinuierliche Verfolgung der Log-Ausgaben. Durch die Eingabe der Tastenkombination [Strg]+[c] (Windows) bzw. [Strg] + [z] (Linux) kann dieser Modus wieder beendet werden.

`--tail <ANZAHL>`
Mit diesem Parameter wird bestimmt, wie viele Zeilen am Ende des Logs ausgeben werden. Vorgabe Wert ist `all`

`--timestamp oder, -t`
Wird diese Option angegeben, so wird zu Beginn eines jeden Log-Eintrags ein Zeitstempel angezeigt. Dieser folgt dem RFC3339Nano-Format
`[YYYY]-[MM]-[DD]T[hh]:[mm]:[ss].[nnnnnnnnn]Z`
Beispiel:
2020-02-04T10:34:51.238007900Z

`--since`
Anzeige der Log-Ausgaben seit einem bestimmten Zeitpunkt (Angabe als Timestamp) oder relativ zum aktuellen Zeitpunkt (z.B. 15m für die letzten 15 Minuten).

`--until`
Anzeige der Log-Ausgaben vor einem bestimmten Zeitpunkt (Angabe als Timestamp) oder relativ zum aktuellen Zeitpunkt (z.B. 15m für die letzten 15 Minuten).

Um die Log-Informationen eines Services auszugeben, gibt es das Kommando `docker service logs`. Das wird aber nicht hier, sondern später in den weiterführenden Kapiteln behandelt.

**Achtung!**

Die Funktionalität des `docker logs`-Kommandos ist nur sichergestellt, wenn Container mit den Logging Treibern „`json-file`“ oder „`journald`“ gestartet worden sind.

Wenn in der Docker-Konfigurationsdatei „`daemon.json`“ nichts anderes angegeben ist, dann wird der Logging-Treiber „`json-file`“ als Default-Treiber aktiv (siehe Kapitel 9.4).

## 9.2 Praktisches Beispiel zur Anzeige der Container Logs

Um die Arbeit mit dem `docker logs`-Kommando praktisch zu demonstrieren, nehmen wir unser Image mit dem PHP-Beispiel "telefonapp" (siehe Kapitel 4.9).

Wir starten mit diesem Image einen Docker Container. Der bekommt, wie gehabt, den Namen „telefon_app". Auch verwenden wir wieder den Parameter `-d`, um den Container im Hintergrund auszuführen. Der interne Port 80 wird auf dem externen Port 8808 veröffentlicht ( `-p 8808:80`).

```
> docker run --name telefon_app -d -p 8808:80 '
<DOCKER_ID>/telefon-app
```

Um für diesen Container die Log-Daten auszulesen, geben wir in der Shell dieses Kommando ein:

```
> docker logs telefon_app
```

Der folgende Screenshot zeigt das Ergebnis der Log-Ausgabe (Abb. 9.1).

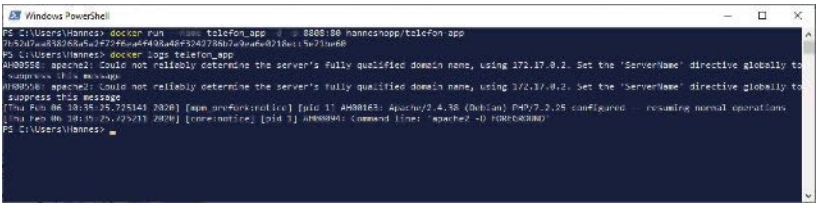



**Abb. 9.1** Erstes Beispiel zur Log-Ausgabe des Containers `telefon_app`

Wir stoppen den Container wieder und fragen noch einmal die Log-Informationen ab:

```
> docker stop telefon_app
> docker logs telefon_app
```

Wie Sie sehen, kann man die Log-Informationen immer noch auslesen. Es ist zur obigen Ausgabe noch ein Eintrag hinzugekommen. Dieser wurde durch die Eingabe des `stop`-Kommandos erzeugt (Abb. 9.2).

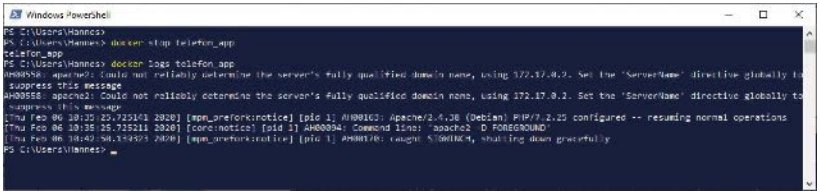



**Abb. 9.2** Zweites Beispiel zur Log-Ausgabe des Containers `telefon_app` nach dem Stoppen

Bisher haben wir zur Kontrolle von Docker Containern nur die beiden Docker-Kommandos `docker run` und `docker stop` kennengelernt. Diese Liste möchte ich nun um ein weiteres Kommando erweitern, welches die Möglichkeit bietet, einen Container, der gestoppt wurde, wieder neu laufen zu lassen. Dieses Kommando heißt `docker restart`. Es kann allerdings nur dann erfolgreich ausgeführt werden, wenn der Container nach dem Stoppen nicht noch zusätzlich mit dem Kommando `docker container rm <CONTAINER_NAME>` gelöscht wurde.



Hier die Syntax für dieses Kommando. Als Option kann nur `--time` bzw. `-t` angegeben werden. Diese Option bestimmt, wie viele Sekunden gewartet werden muss, bevor der Container zerstört wird:

```
> docker restart [OPTIONS] <CONTAINER_NAME> [<CONTAINER_NAME>]
```

Mit dem folgenden Kommando starten wir den Container unserer Telefon-App, den wir vorhin gestoppt haben, wieder neu, stoppen ihn noch einmal und betrachten die neuen Einträge in der Log-Datei (Abb. 9.3).

```
> docker restart telefon_app
> docker stop telefon_app
> docker logs telefon_app
```

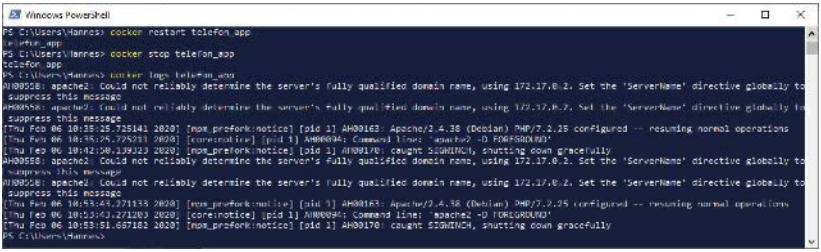



**Abb. 9.3** Drittes Beispiel zur Log-Ausgabe des Containers `telefon_app` nach `restart` und `stopp`.

Mann kann hier erkennen, dass das Container Logfile durch die Informationen beim Neustart und beim Stoppen ergänzt wurde.

Aber jetzt löschen wir den Container und versuchen dann die Log-Daten anzuzeigen (Abb. 9.4).

```
1  > docker container rm telefon_app
2  > docker logs telefon_app
```

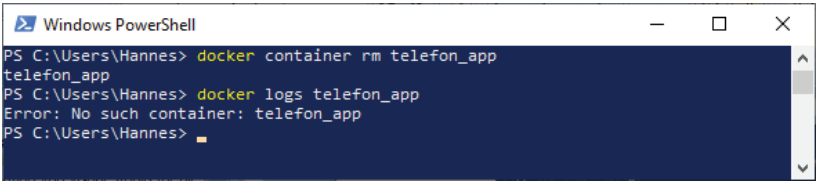



**Abb. 9.4** Viertes Beispiel zur Log-Ausgabe des Containers `telefon_app` nach dem Löschen des Containers

Wie bereits im Kapitel 4.7.3 angesprochen wurde, verschwindet ein Container-Dateisystem erst mit dem Löschen dieses Containers. Die Log-Dateien befinden sich ebenfalls im Container-Dateisystem. Nachdem der Container durch das Kommando `docker container rm` entfernt wurde, sind auch seine Log-Informationen unwiederbringlich verloren.

Nach einem Neustart eines Containers mit demselben Image sind dann auch nur die Log-Informationen ab dem Start dieser neuen Containerinstanz verfügbar (Abb. 9.5).

```
> docker run --name telefon_app -d -p 8808:80 <DOCKER_ID>/
telefon-app
> docker logs telefon_app
```

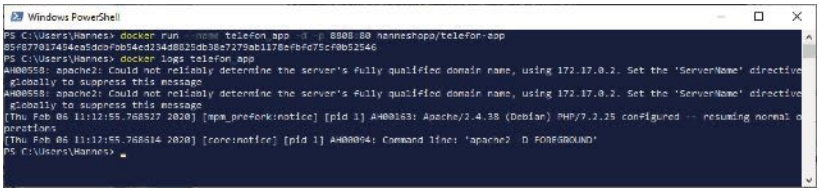



**Abb. 9.5** Fünftes Beispiel zur Log-Ausgabe des Containers `telefon_app` nach dem Neustart des Containers



## 9.3 Kontinuierliche Log-Ausgaben

Falls Sie Log-Ausgaben während der Laufzeit in Echtzeit mit verfolgen wollen, dann bietet das `docker logs`-Kommando diese Funktion durch den Parameter `-f`.

Probieren wir es aus. Nach dem Start des Containers „telefon_app“ rufen wir das Kommando `docker logs` mit dem Parameter `-f` auf:

```
> docker logs -f telefon_app
```

Das Docker-Kommando kehrt bei einem solchen Aufruf nicht zur Shell zurück, sondern läuft im Hintergrund weiter.

Starten Sie jetzt einen Web-Browser und geben in der Adresszeile die URI für den „localhost“ mit der Portnummer 8808 an, um die Webseite von „telefon_app“ anzuzeigen.

```
localhost:8808
```

Wenn wir jetzt auf dieser Seite nacheinander Namen eingeben und den Button [Suchen] klicken, dann können Sie die Log-Ausgaben, die durch diese Aktionen erzeugt werden, direkt mitlesen.

Wollen Sie diesen Modus wieder stoppen und zur Shell zurückkehren, dann betätigen Sie die Tastenkombination [Strg] + [c]. wenn Sie mit der Windows PowerShell arbeiten. Unter Linux verwenden Sie dagegen die Tastenkombination [Strg] + [z].

Der folgende Screenshot zeigt ein Beispiel für die kontinuierlichen Log-Ausgaben aus diesem Beispiel (Abb. 9.6).


**Abb. 9.6** Sechstes Beispiel zur Log-Ausgabe des Containers `telefon_app` mit parameter -f

## 9.4 Logging-Treiber konfigurieren

Docker bietet verschiedene Logging-Dienste an, mit deren Hilfe man Informationen aus laufenden Docker Containern und Services erhalten kann. Man nennt diese Dienste Logging Driver.

Für jeden Container ist ein Standard-Treiber eingerichtet, über den Logging-Informationen zur Verfügung gestellt werden. Sie können aber die Konfiguration ändern, falls Sie einen anderen Logging-Treiber als Standard spezifizieren wollen.

Darüber hinaus ist man nicht auf die Logging-Treiber beschränkt, welche in Docker standardmäßig integriert sind. Man kann andere Logging-Treiber über einen Plug-In-Mechanismus in Container einbinden.

Wir wollen uns hier aber zunächst nur mit den Logging-Treibern befassen, die von Docker bereitgestellt werden.

### 9.4.1 Konfiguration des Standard-Logging-Treibers

Als Standard-Logging-Treiber wird von Docker ein Treiber mit dem Namen „`json-file`“ genutzt. Dieser Treiber fängt die Ausgaben der Standard-Ausgabegeräte `stdout` und `stderr` ab und überträgt sie im JSON-Format in Log-Dateien.

Wollen Sie einen anderen Logging-Treiber verwenden, dann können Sie das durch die Bearbeitung einer speziellen Konfigurationsdatei erreichen. Diese Datei hat den Dateinamen „`daemon.json`“ und befindet sich auf Windows-PCs im Verzeichnis „`C:\ProgramData\Docker\config`“.



Es kann sein, dass auf Ihrem Computer für das Verzeichnis C:\`ProgrammData` das Attribut ‚hidden‘ gesetzt ist und der Datei-Explorer das Verzeichnis deswegen nicht anzeigt. Wenn Sie aber in der Eingabezeile des Datei-Explorers den obigen Pfad direkt eingeben, dann wechseln Sie in das Verzeichnis und können die Datei „`daemon.json`“ von dort öffnen und bei Bedarf bearbeiten.

Auf Linux-Systemen ist die Konfigurationsdatei „`daemon.json`“ im Verzeichnis „`/etc/docker/`“ abgelegt.

Der Eintrag für den Logging-Treiber hat für diese Datei das folgende Format:

```
{
  "log-driver": "<driver>"
  ["log-opts":
  {
    "<option>": "<value>",
    […]
```

```
  }]
}
```

Wenn zum Beispiel als Default Logging-Treiber „`syslog`“ verwendet werden soll, dann muss in der Datei „`daemon.json`“ der folgende Eintrag vorhanden sein:

```
Datei ,daemon.json'
…
{
  "log-driver": "syslog"
}
…
```

Die folgende Tabelle enthält eine Auswahl von verfügbaren Logging-Treibern unter Docker:

| ***Name*** | ***Beschreibung*** |
|---|---|
| none | Keine Verwendung von Logs |
| json-file | Die Log-Dateien werden im JSON-Format geführt. Das ist die Standardeinstellung von Docker. |
| syslog | Der syslog-Treiber routet die Log-Ausgaben zu einem syslog-Server. Dazu muss der syslog-Daemon aktiviert sein. |
| journald | Der `journald` Logging-Treiber sendet die Log-Ausgaben zum `systemd`-Journal. Dazu muss der `journald`-Daemon auf dem Host laufen sein. Log-Ausgaben können durch das `journalctl`-Kommando abgerufen werden oder durch das `docker logs`-Kommando. |
| awslogs | Log Messages werden an „Amazon CloudWatch Logs“ gesendet. Log-Ausgaben können unter anderem über die „AWS Managemanent Console“ ausgelesen werden. |

| | |
|---|---|
| etwlogs | Log Messages werden als „Events Tracing für Windows“ (ETW) erstellt. Dieser Logging-Treiber ist nur auf Windows-Plattformen verfügbar. |
| gcplogs | Dieser Treiber sendet Log Messages zum Google Cloud Platform (GCP) Logging. |

Bei Bedarf kann der Eintrag für den Logging-Treiber um Konfigurations-Optionen erweitert werden. Dazu wird der Eintrag um ein JSON-Objekt mit dem Key-Namen „log-opts“ ergänzt. Innerhalb dieses Objekts werden die Optionen als Key-Value-Paare aufgelistet. Sind mehrere Optionen vorhanden, so trennt man diese durch Komma. Nach dem letzten Element, vor der schließenden geschweiften Klammer, darf kein Komma gesetzt werden. Bitte beachten Sie, dass hier auch Zahlenwerte wie eine Zeichenkette in Anführungszeichen angegeben werden müssen.



Hier ein Beispiel für den Treiber „`json-file`“ mit der Angabe von zusätzlichen Optionen.

```
Datei ‚daemon.json‘
…
{
  "log-driver": "json-file",
  "log-opts":
  {
    "max-size": "100k",
    "max-file": "2",
    "labels": "production_status",
  }
}
…
```

Die folgende Tabelle zeigt die verfügbaren Optionen für den Treiber „`json-file`“:

| *Option* | *Beschreibung* |
|---|---|
| max-size | Diese Option bestimmt die maximale Größe der Log-Datei. Wird dieser Wert überschritten, dann werden die ältesten Einträge im Log gelöscht. Als Wert wird ein positiver Integer-Wert mit einem Kürzel für die Maßeinheit angegeben (k – Kilo, m – Mega, g – Giga). Der Wert -1 bedeutet unbegrenzt (Standard). |
| max-file | Die maximale Anzahl von Log-Dateien, die vorhanden sein können. Auch hier werden die ältesten Dateien entfernt, wenn dieser Wert überschritten wird. Die Angabe wirkt nur in Kombination mit max-size. Ohne Angabe wird als Standardwert 1 gesetzt. |
| labels | Wird zur Erstellung von erweiterten Log-Einträgen angegeben. Als Wert wird eine kommaseparierte Liste mit Logging relevanten Labels angegeben.<br>Beispiel:`labels=production_status,geo` |
| env | Wird zur Erstellung von erweiterten Log-Einträgen angeben. Als Wert wird eine kommaseparierte Liste mit Logging relevanten Environment-Variablen angegeben. |
| compress | Aktiviert die Komprimierung beim Einsatz von Log-Rotation (Logrotate). Die Komprimierung ist standardmäßig deaktiviert (`false`).<br>Beispiel: `compress=true` |

Beachten Sie, dass die Treiber-Optionen dieser Tabelle nur für den Treiber „`json-file`“ gültig sind. Jeder Treiber hat seine eigenen Optionen. Diese können Sie in der Online-Dokumentation für Docker nachlesen.

Ein guter Einstiegspunkt dafür ist die folgende Web-Adresse:

https://docs.docker.com/config/containers/logging/configure/

### 9.4.2 Konfiguration des Logging-Treibers für einen Container

Wollen Sie für einen Docker Container vorübergehend einen anderen Log-Treiber als den standardmäßig vorgegebenen verwenden, dann können Sie dies beim Start des Containers per Parameter bestimmen. Dafür gibt es für das Container `run`-Kommando das Flag `--log-driver`.

Das folgende Kommando startet unsere „Telefon-App“ als Container ohne Log-Treiber

```
> docker run --name telefon_app --log-driver none '
-d -p 8808:80 hanneshopp/telefon-app
```

Der nächste Screenshot zeigt den Start eines Containers für das `telefon-app` Image mit dem Log-Treiber „`none`“, also ohne Log-Treiber. Danach wird versucht, die Log-Ausgaben für diesen Container zu erhalten. Der dritte Befehl des Screenshots zeigt, wie man mit dem Kommando `docker inspect` den aktuell aktiven Log-Treiber abfragen kann (Abb. 9.7). Dem Parameter -f übergibt man einen Formatstring im Go-Format. Der Anwendung dieses Formats wird im Anhang Kapitel *19.11 Format Angaben für Docker-Kommandos* beschrieben.

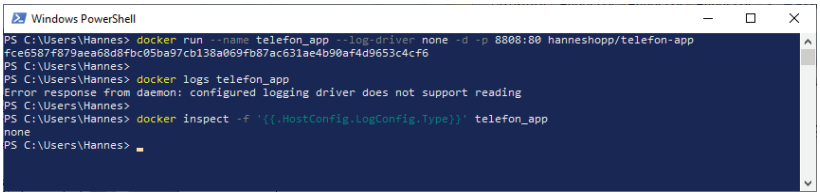



**Abb. 9.7** Start des Containers `telefon_app` ohne Log-Treiber

## 9.5 Container Logs persistent auslagern

Die Log-Dateien eines Containers werden im Container-Dateisystem gespeichert. Ohne besondere Maßnahmen sind Log-Dateien eines Containers nur solange vorhanden, bis der Container mit dem Kommando `docker container rm` gelöscht wird. Wenn sich das Verzeichnis mit

den Log-Dateien allerdings persistent in einem Docker Volume befindet, dann sind dessen Log-Daten über die Lebensdauer des Containers hinaus weiter vorhanden.

Wir müssen dazu eigentlich nur wissen, wo im Container-Dateisystem die Log-Dateien gespeichert sind. Auch das ist kein großes Geheimnis – das Verzeichnis der Log Dateien im Container heißt:

```
1 /var/log
```

Starten wir also unseren Docker Container mit der `telefon_app`, geben einen Volume-Namen an und mappen diesen auf das Verzeichnis `/var/logs`.

```
1 > docker run --name telefon_app -d -p 8808:80 '
2 -v logs:/var/log hanneshopp/telefon-app
```

Jetzt lassen wir uns noch einmal die Log-Informationen direkt nach dem Containerstart anzeigen.

```
1 > docker logs telefon_app
```

Wieder starten wir einen Web-Browser und geben in der Adresszeile die URI für den localhost mit der Portnummer 8808 an.

```
1 localhost:8808
```

Wenn unsere Webseite angezeigt wird, geben wir dort einen Namen ein und klicken auf den Button [Suchen].

Wir fragen erneut die Log-Informationen ab und beobachten die Veränderungen.

```
1 > docker logs telefon_app
```

Jetzt wird der Container gestoppt und entfernt.

```
1 > docker stop telefon_app
2 > docker container rm telefon_app
```

Und wieder gestartet ohne die Angabe eines Volumes:

```
> docker run --name telefon_app -d -p 8808:80 '
hanneshopp/telefon-app
```

Prüfen Sie danach wieder die Log-Ausgaben. Es sind deutlich weniger Einträge als vorhin.

Den Container nochmals stoppen und entfernen. Es folgt ein erneuter Start mit der Angabe des Volumes.

```
> docker stop telefon_app
> docker container rm telefon_app
> docker run --name telefon_app -d -p 8808:80 '
-v logs:/var/log hanneshopp/telefon-app
```

Jetzt sehen wir uns erneut die Log-Ausgaben an, ohne vorher auf der Webseite irgendwelche Aktionen durchzuführen.



```
> docker logs telefon_app
```

Die Log-Einträge der vorigen Container-Instanz sind wieder da und werden noch durch weitere, neu erstellte Log-Informationen ergänzt.

Kapitel 10

# Netzwerke und Docker

Wie schon im Einführungskapitel (Kapitel 2.6) erwähnt, sind in Docker standardmäßig drei Netzwerke auf dem Docker Host installiert – *none*, *bridge* und *host*.

Netzwerkfunktionalität wird unter Docker durch Treiber realisiert. Die Treiber für *bridge*- und *host*-Netzwerke sind vorinstalliert und bieten Basisfunktionalitäten für die Netzwerkkommunikation.

Wir wollen uns in diesem Kapitel genauer mit den Standardnetzwerken befassen. Dann sehen wir uns an, wie man eigene Netzwerke für Docker Container erstellt und einrichtet.



Betrachten wir zunächst die aktuell eingerichteten Netzwerke. Docker stellt uns dafür das folgende Kommando zur Verfügung:

```
1  > docker network ls
```

Starten wir also eine Shell und geben dieses Kommando einmal ein (Abb. 10.1).

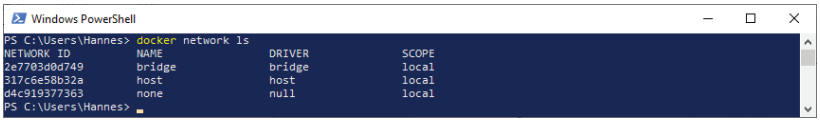



**Abb. 10.1** Docker Netzwerke anzeigen

Wenn Sie bisher keine eigenen Netzwerke angelegt haben, dann sollte die Ausgabe so ähnlich wie im obigen Screenshot aussehen. Es werden hier nur Informationen über die drei Standard-Netzwerke von Docker angezeigt.

Um detaillierte Informationen über ein Netzwerk zu erhalten, verwenden wir das Kommando `docker network inspect`.

Hier die Syntax:

```
> docker network inspect [<OPTIONS>]<NETWORK> [<NETWORK> …]
```

Folgende Optionen stehen für dieses Kommando zur Verfügung:

```
-v, --verbose
```

Detaillierte Ausgabe für Diagnosezwecke.

```
-f, --format <STRING>
```

Formatangabe für die Ausgabe des Kommandos.

Die Struktur des Format-Strings wird im Anhang Kapitel *19.11 Format Angaben für Docker-Kommandos* beschrieben.

### 10.1.1 None

Genau genommen ist das Netzwerk *none* gar kein Netzwerk. Damit gibt man lediglich an, dass ein Container mit keinem Netzwerk verbunden werden soll. Die Netzwerkfunktion ist damit vollständig deaktiviert.

Sehen wir uns die Informationen an, die uns Docker durch das Kommando `docker network inspect` für den Netzwerktype *none* zur Verfügung stellt (Abb. 10.2).

Dafür geben wir das Kommando wie folgt in einer Shell ein:

```
> docker network inspect none
```

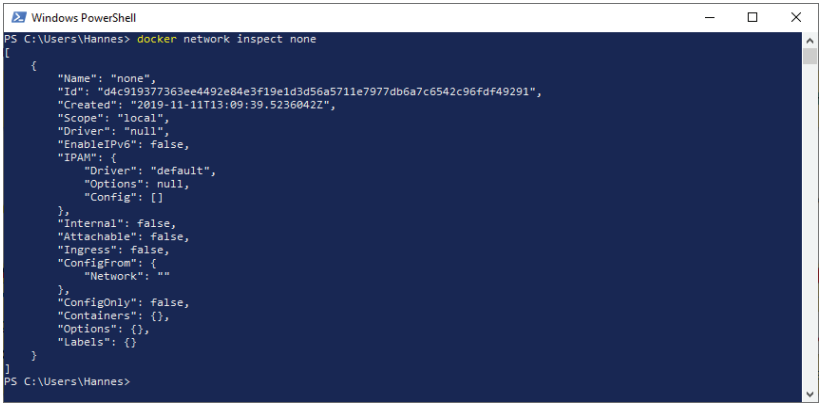



**Abb. 10.2** Informationen zum Docker-Netzwerk none

Im Abschnitt „IPAM“ (IP Adress Management) kann man sehen, dass für „Config“ weder eine IP für das Subnet, noch eine IP für das Gateway eingetragen ist.



Als Subnet oder Subnetz versteht man ein Teilnetz aus einem übergeordneten Netz. Hat ein übergeordnetes Netz zum Beispiel die IP-Adresse 192.168.0.0/16, dann wäre die Adresse 192.168.178.0/24 ein Subnetz in diesem übergeordneten Netz. Durch den Wert nach dem Schrägstrich wird bestimmt, wie viele Bits der IP-Adresse die Netzwerkadresse bestimmen. Man nennt das die Subnet-Maske.

| | IP-Adresse Dezimal | IP-Adresse Binär |
|---|---|---|
| IP-Adresse | 192.168.0.4/16 | 11000000.10101000.<br>00000000.00000100 |
| && Subnet-Maske | 255.255.0.0 | 11111111.11111111.<br>00000000.00000000 |
| Subnet-Adresse | 192.168.0.0 | 11000000.10101000.<br>00000000.00000000 |

| | IP-Adresse Dezimal | IP-Adresse Binär |
|---|---|---|
| IP-Adresse | 192.168.178.6/24 | 11000000.10101000.10110010.00000110 |
| && Subnet-Maske | 255.255.255.0 | 11111111.11111111.11111111.00000000 |
| Subnet-Adresse | 192.168.178.0/24 | 11000000.10101000.10110010.00000000 |

Als *Gateway* bezeichnet man eine Hard- oder auch Softwarekomponente, welche die Verbindung zu anderen Systemen außerhalb herstellt. Das *default Gateway* in einer IP-Konfiguration leitet alle nicht zu einem Subnetz gehörenden Netzwerkanfragen an ein anderes Subnetz weiter.

Wenn Sie z.B. eine Anfrage zu einer URL im Internet senden wollen, dann wird diese zunächst an die Adresse des Gateways geschickt. Dieses leitet die Anfrage dann über das Internet weiter an die zugehörige Zieladresse und sorgt im Anschluss dafür, dass auch die Antwort wieder an den richtigen Adressaten zurückgereicht wird.

### 10.1.2 Host

Das `host`-Netzwerk verbindet einen Container mit dem hosteigenen Netzwerk. Dadurch wird ein Container direkt an die IP-Adresse des Container Hosts angebunden.

Der `host`-Netzwerktreiber steht aber nur auf Linux Hosts zur Verfügung. Er wird von den Applikationen *Docker Desktop für Mac*, *Docker Desktop für Windows* und *Docker EE für Windows Server* nicht unterstützt.

Der Docker-Netzwerktyp *host* eignet sich im Grunde genommen auch nur für Container, die alleine auf einem Linux-Host-Rechner ausgeführt werden.

Ein großer Nachteil ist hierbei, dass die Isolation der Netzwerke von Host und Container aufgehoben wird. Das bedeutet weniger Schutz

und Sicherheit sowohl für den Host als auch für den Container, denn der Netzwerk Stack des Hosts wird im Container integriert. Container, die im *host*-Modus laufen, haben damit den vollen Zugriff auf die Host-Schnittstellen und könnten dort Schaden anrichten.

Der Vorteil von Containern, die im *host*-Netzwerkmodus arbeiten, ist der, dass sie eine höhere Performance bieten können, als dies im zum Beispiel im *bridge*-Modus möglich ist.

Sinnvoll kann der *host*-Modus auch dann sein, wenn von einem Container eine große Anzahl von Ports zu handhaben ist. Dabei darf dann aber keine NAT (Network Address Translation) Funktionalität für die Ports erforderlich sein.

Jetzt sehen wir uns mit dem Kommando `docker network inspect` die Informationen für den Netzwerktype *host* an (Abb. 10.3).

Dafür geben wir das Kommando wie folgt in einer Shell ein:



```
1  > docker network inspect host
```

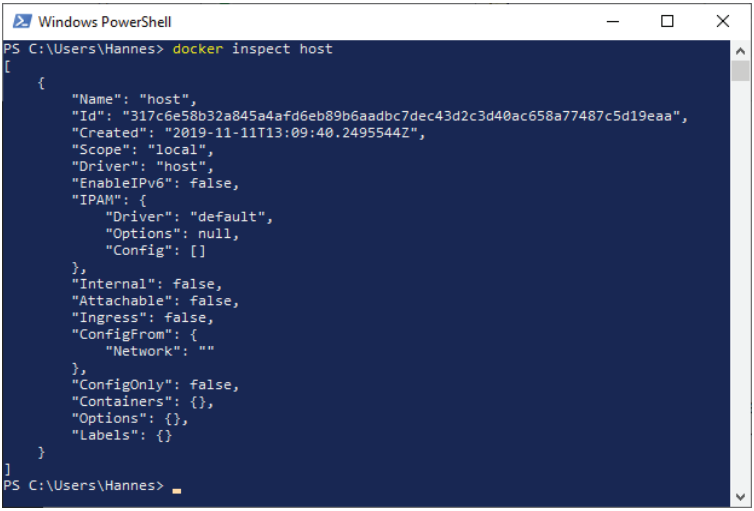



**Abb. 10.3** Informationen zum Docker-Netzwerk host

### 10.1.3 Bridge

Das *bridge*-Netzwerk ist das Standard-Netzwerk von Docker. Beim Start von Docker wird automatisch das *bridge*-Netzwerk aktiviert. Startet ein Container, so wird dieser, falls kein anderes Netzwerk angegeben wird, mit dem *bridge*-Netzwerk verbunden.

Das Docker *bridge*-Netzwerk verwendet eine Software Bridge, welche die Netzwerk-Segmente verschiedener Container miteinander verbindet. Die Container müssen aber bei *bridge*-Netzwerken auf dem gleichen Host-Rechner ausgeführt werden.

*Bridge*-Netzwerke ermöglichen die Kommunikation von Containern, die dem gleichen Netzwerk angehören. Gleichzeitig werden Container voneinander isoliert, wenn sie sich nicht im gleichen Netzwerk befinden.

Fragen wir auch hier für das *bridge*-Netzwerk die Informationen ab, die uns Docker bereitstellt (Abb. 10.4).

Dafür geben wir das Kommando wie folgt in einer Shell ein:

```
1  > docker network inspect bridge
```

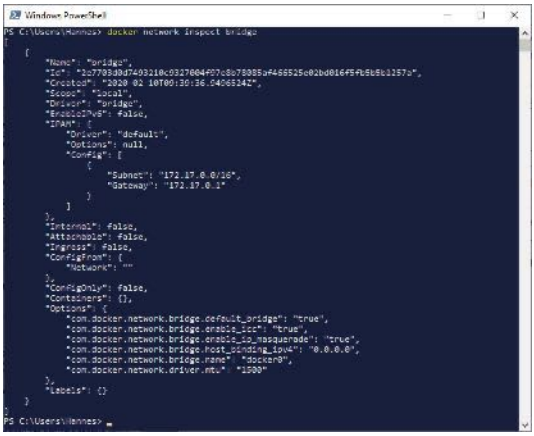



**Abb. 10.4** Informationen zum Docker-Netzwerk bridge

Wie Sie hier sehen können, wurde für das bridge-Netzwerk der Sub-Netz-Adressbereich [172.17.0.0/16] als Standard vergeben. Die Netzmaske hat somit den Wert [255.255.0.0] und es gibt maximal 65.534 nutzbare IP-Adressen, nämlich von [172.17.0.0.2] bis [172.17.0.0.254]. Die Adresse des Application Layer Gateways ist [127.17.0.1]. Das repräsentiert die Adresse der SPI (Stateful Packet Inspection) Firewall im Host-Rechner.

Man kann aus der Anzeige auch ersehen, dass das *bridge*-Netzwerk für Docker das Standard-Netzwerk-Interface mit dem Namen „docker0" bereitstellt.

Das Standard-*bridge*-Netzwerk wird mittlerweile als eine veraltete Funktionalität von Docker betrachtet. Die Nutzung im Produktiv-Bereich wird aus diesem Grund nicht mehr empfohlen.

Stattdessen sollten Benutzerdefinierte bridge-Netzwerke zum Einsatz kommen.



### 10.1.4 Benutzerdefinierte bridge-Netzwerke

Zum Anlegen benutzerdefinierter Netzwerke verwenden wir das Kommando `docker network create` mit dieser Syntax:

```
1  > docker network create [<OPTIONEN>] <NAME>
```

Außer zahlreichen anderen Optionen ist die für uns wichtigste der Parameter --driver bzw. -d, über den der zu verwendende Netzwerk-Treiber angegeben werden kann. Standard-Vorgabewert ist dabei der Treiber *bridge*. Darum wird es in diesem Kapitel gehen.

Wir erstellen jetzt unser eigenes *bridge*-Netzwerk, das den Namen `test_net` erhalten soll.

```
1  > docker network create -d bridge test_net
```

Da als Standard-Netzwerk-Treiber, wie oben bereits erwähnt, sowieso der *bridge*-Treiber verwendet wird, kann man die Option `-d` auch ohne Weiteres weglassen.

```
> docker network create telefon_net
```

Sehen wir uns im Anschluss die neue Liste der aktuell verfügbaren Netzwerke an (Abb. 10.5).

```
> docker network ls
```

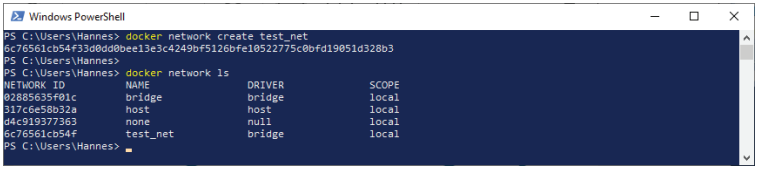



**Abb. 10.5** Anzeige der Docker-Netzwerkliste mit benutzerdefiniertem Netzwerk.

Jetzt taucht in dieser Liste unser neues Netzwerk als viertes Element der Liste auf. Die Spalte DRIVER zeigt uns außerdem, dass das neue Netzwerk von Typ *bridge* ist.

Noch mehr Informationen über unser neues benutzerdefiniertes Netz erhalten wir wieder mit dem Inspect-Kommando (Abb. 10.6).

```
> docker network inspect test_net
```

**Abb. 10.6** Informationen zum benutzerdefinierten Docker-Netzwerk

Was sich hier jetzt verändert hat, sind die IP-Adressen für *Subnet* und *Gateway*. Aus der Subnet-Adresse des Standard-*bridge*-Netzwerks [172.17.0.0/16] wurde [172.18.0.0/16] und aus der Gateway-Adresse [172.17.0.1] wurde [172.18.0.1].

Jedes Mal, wenn ein neues Netzwerk in Docker angelegt wird, erhält es eine Netzwerk-Adresse, bei der die zweite Stelle der IP-Adresse um eins erhöht wird.

### 10.1.5 Overlay

Mit dem *overlay*-Treiber können Netzwerke erstellt werden, die Docker Container verbinden, welche auf mehreren Host-Rechnern verteilt sind. *Overlay*-Netzwerke setzen auf die hostinternen Netzwerke auf und erlauben eine sichere Kommunikation zwischen Containern. Docker selbst kümmert sich um das Routing der Pakete zwischen Host-Rechnern und deren Containern.



Bei der Arbeit mit Docker Swarm kommen vordefinierte *overlay*-Netze zum Einsatz. Mehr dazu folgt in den Kapiteln zum Thema Docker Swarm.

Benutzerdefinierte *overlay*-Netzwerke werden auf die gleiche Art und Weise erstellt wie benutzerdefinierte bridge-Netzwerke. Als Netzwerk-Treiber muss über den Parameter `-d` jetzt eben *overlay* angegeben werden.

```
1  > docker network create -d overlay <NETZWERK_NAME>
```

Das Kommando funktioniert nur für den Einsatz mit Swarm Services. Der Docker Daemon muss dafür als Swarm Manager initialisiert worden sein.

### 10.1.6 Macvlan

Für den Fall, dass Applikationen direkten Zugriff auf die physikalische Schicht eines Netzwerks benötigen, wird von Docker der Netzwerktyp *macvlan* bereitgestellt. Dieser Netzwerktyp erlaubt es, der Netzwerkschnittstelle eines Containers MAC (Media Access Control) Adressen zuzuweisen, um so eine physikalische Schnittstelle zum Netzwerk zu simulieren.

Da diese Funktionalität nur in wenigen speziellen Sonderfällen benötigt wird (zum Beispiel zur Anbindung älterer Anwendungen oder Anwendungen zur Netzwerküberwachung), gehen wir an dieser Stelle nicht näher darauf ein.

### 10.1.7 Container mit Netzwerk verbinden

Soll ein Container über ein Netzwerk kommunizieren, so muss er mit diesem Netzwerk verbunden sein. Die Netzwerkanbindung eines Containers kann sowohl bei Container-Start erfolgen als auch zur Laufzeit, wenn ein Container bereits ausgeführt wird.

Beim Start eines Containers kann man den Namen des Netzwerks, an welches der Container angebunden werden soll, durch einen Kommandozeilen-Parameter mit dem Namen `--network` an das `run`-Kommando übergeben. Allerdings ist hier nur die Angabe eines einzelnen Netzwerk-Namens möglich.

Das folgende Beispiel zeigt, wie ein Nginx Container beim Start an das Netzwerk `test_net` angebunden wird. Anschließend sehen wir uns an, wie sich das auf die Netzwerkeigenschaften auswirkt (Abb. 10.7).

```
1 > docker container run -d --name nginx --network test_net nginx
2 > docker network inspect test_net
```

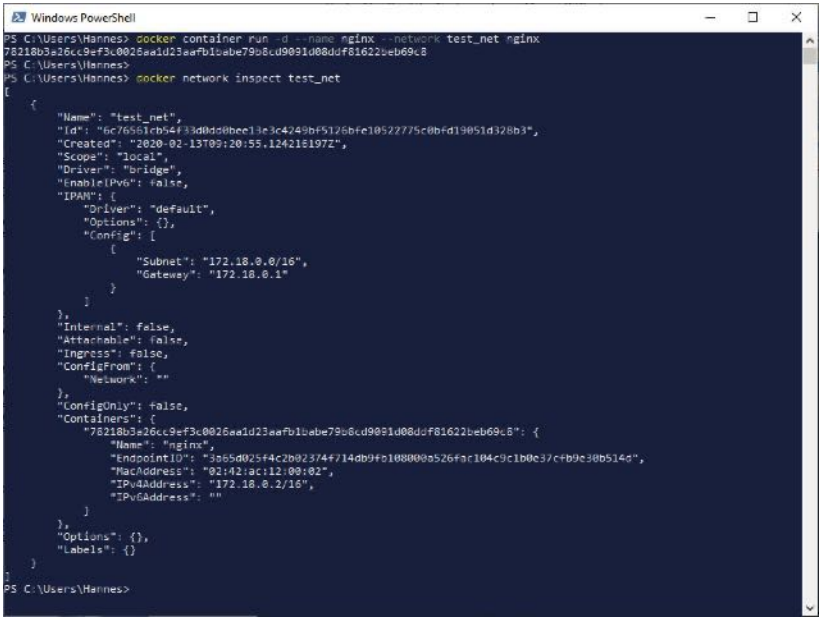



**Abb. 10.7** Container an benutzerdefiniertes Netzwerk anbinden

Bei der Ausgabe der Netzwerkinformationen kann man jetzt sehr schön sehen, dass sich die Sektion *Containers* verändert hat. Dort befinden sich in einem neuen Eintrag die Informationen zum angebundenen Container „nginx“. Für diesen ist hier ist die MAC-Adresse [02:42:ac:12:00:02] und eine neue IP-Adresse [172.18.0.2] zugewiesen worden.

Um zu sehen, was sich durch die Verbindung zum Netzwerk im Docker Container selbst verändert hat, sehen wir uns die Detail-Informationen des Containers mit `inspect` an (Abb. 10.8).

```
1  > docker container insepect nginx
```

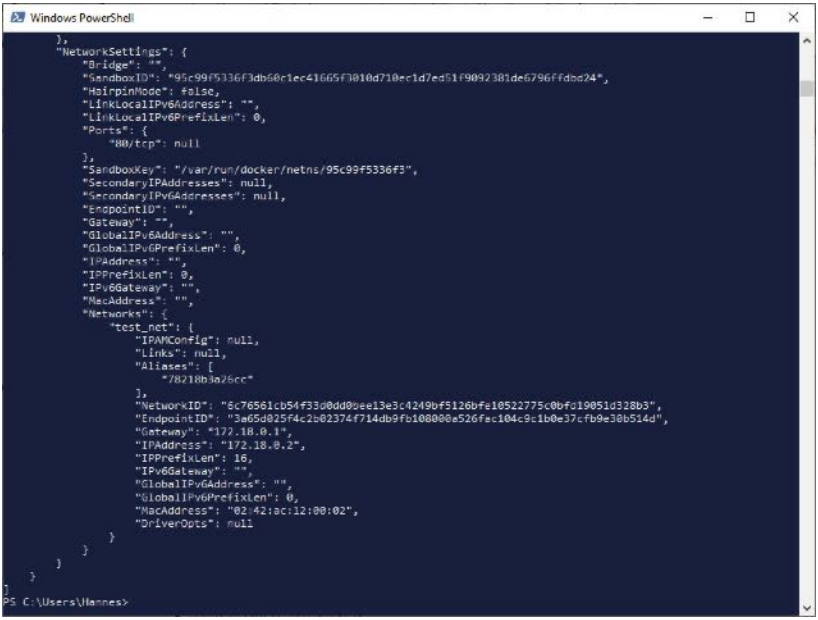



**Abb. 10.8** Informationen eines Containers mit Netzwerkanbindung

Es wird durch dieses Kommando eine lange Liste mit Informationen ausgegeben. Am Ende der Ausgabe erscheinen die Netzwerk-Einstellungen. Dieser Teil ist in obigem Screenshot sichtbar.

Sie können auch hier unter anderem die folgenden Netzwerk-Einstellungen für diesen Container erkennen:

```
"Gateway": "172.18.0.1",
"IPAddress": "172.18.0.2",
"IPPrefixLen": 16,
"MacAddress": "02:42:ac:12:00:02",
```

Wollen wir unseren Container mit weiteren Netzwerken verbinden, dann können wir dies tun, nachdem er gestartet wurde.

Einen laufenden Container verbinden wir mit einem Netzwerke durch Aufruf des Kommandos `docker network connect`.

Hier zunächst die Syntax:

```
> docker network connect [<OPTIONEN>] <NETZWERK> <CONTAINER>
```

Um das auszuprobieren, erstellen wir noch ein Netzwerk mit dem Namen `demo_net`. Auch an dieses Netz wird im Anschluss der Container `nginx` angebunden. Wir sehen uns die Eigenschaften des neuen Netzes dann auch gleich an (Abb. 10.9).

```
> docker network create demo_net
> docker network connect demo_net nginx
> docker network inspect demo_net
```

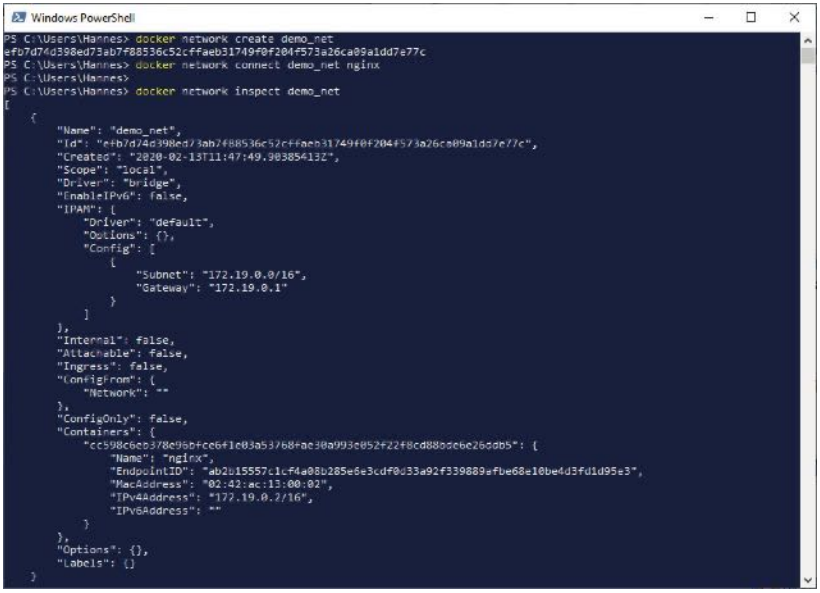



**Abb. 10.9** Laufenden Container an das Netzwerk anbinden

Danach sehen wir uns an, wie das die Eigenschaften des Containers beeinflusst hat (Abb. 10.10):

```
> docker container insepect nginx
```

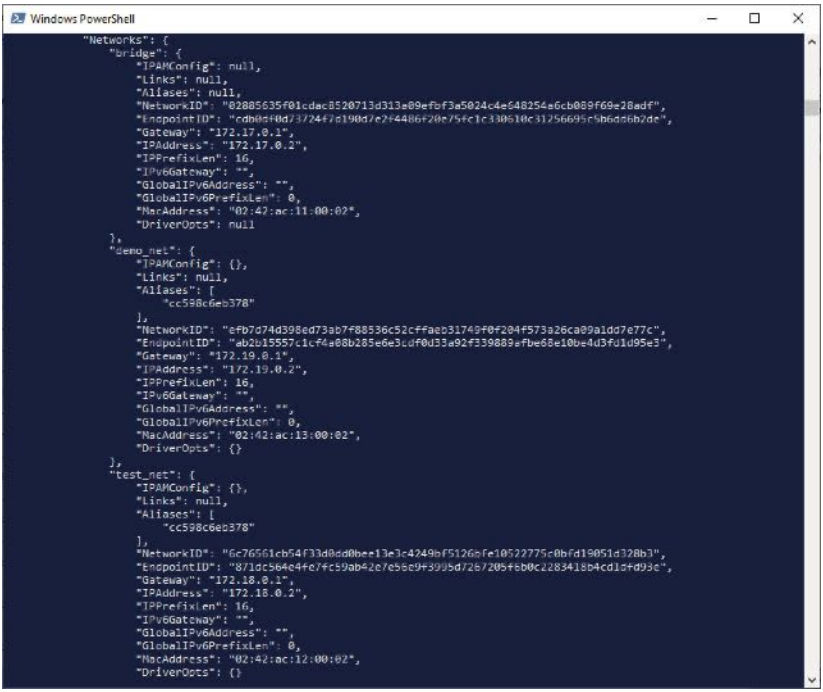



**Abb. 10.10** nginx Container mit drei Netzwerken

Der Screenshot zeigt die letzten Ausgabe-Zeilen mit den Netzwerkinformationen, die durch das Kommando `docker container inspect` erzeugt wurden.

Der Container ist demnach mit drei Netzwerken verbunden.

*bridge*:
Diese Netzwerkanbindung wird ohne besondere Angaben beim Start eines Containers als Standard eingerichtet. In diesem Netzwerk hat der Container die IP [172.17.0.2].

*demo_net*:
Diese Netzwerkanbindung haben wir durch das Kommando `docker network connect` bei bereits gestartetem Container eingerichtet. In diesem Netzwerk hat der Container die IP [172.19.0.2].

*test_net*:
Diese Netzwerkanbindung haben wir beim Start des Containers über den Parameter `–network` definiert.
In diesem Netzwerk hat der Container die IP [172.18.0.2].

Wenn Sie einen bereits gestarteten Container an ein Netzwerk anbinden wollen, dann muss der Container zwar existieren, er kann dabei aber durchaus in der Zwischenzeit gestoppt worden sein.

Nach einem Restart des Containers werden für diesen alle Netzwerkanbindungn etabliert. Diese werden dann auch bei Aufruf des Kommandos `docker network inspect` in der Liste der verbundenen Container angezeigt.



### 10.1.8 Container von einem Netzwerk entfernen

Container werden beim Löschen mit dem Kommando `docker container rm` automatisch von allen verbundenen Netzwerken getrennt.

Laufende Container kann man durch den Parameter `diconnect` des `docker network`-Kommandos von einem verbundenen Netzwerk wieder trennen. Die Syntax sieht folgendermaßen aus:

```
1  > docker network disconnect [<OPTIONS>] <NETZWERK> <CONTAINER>
```

Hier ein Beispiel, um unseren Container `nginx` vom Netzwerk `test_net` zu trennen. Anschließend sehen wir uns das Ergebnis mit dem `network inspect`-Kommando an (Abb. 10.11).

```
1  > docker network disconnect test_net nginx
```

```
2  > docker network inspect test_net
```

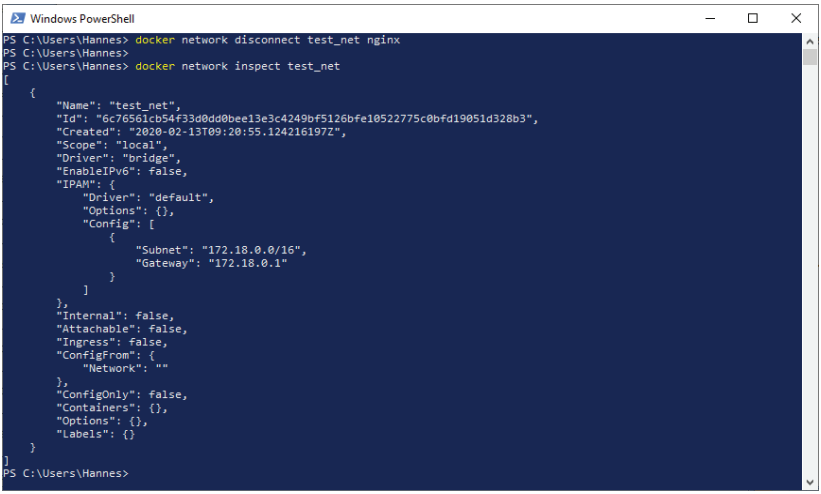



**Abb. 10.11** Container von einem Netzwerk entfernen

### 10.1.9 Übungsaufgabe: Arbeit mit Docker-Netzwerken

Um die Informationen zu Docker-Netzwerken weiter zu vertiefen, stellen wir noch einige praktische Übungen zum Mitmachen vor.

Zuerst erstellen Sie einen neuen Docker Container auf der Basis des Alpine Images. Es handelt sich hier um ein recht kleines Image, das auf Alpine Linux basiert. Geben Sie dem Container den Namen `my_alpine`, verbinden ihn mit dem oben erstellten Netzwerk `demo_net` und geben Sie mit an, dass die Shell ash gestartet werden soll.

***Lösung:***

```
1  > docker run -dit --name my_alpine --network demo_net alpine ash
```

Der Docker Container mit dem Namen `nginx` sollte noch laufen und er sollte mit dem Netzwerk `demo_net` verbunden sein. Falls nicht, dann starten Sie ihn wieder. Finden Sie heraus, welche IP-Adresse der `nginx` Container im Netzwerk `demo_net` besitzt.

***Lösung:***

```
> docker container run -d --name nginx --network demo_net nginx
> docker network inspect demo_net
```


**Abb. 10.12** Erster Screenshot zur Netzwerk-Übungsaufgabe

Bei diesem Screenshot ist das die IP [172.19.0.2].

Um den lokalen Standard-input, output und error streams mit einem laufenden Container verbinden gibt es das Attach-Kommando mit folgender Syntax:

```
> docker container attach [<OPTIONEN>] <CONTAINER>
```

Stellen Sie die Verbindung zum Container `my_alpine` her. Wenn das geklappt hat, sehen Sie den System Prompt der Shell `ash` (#).

Führen Sie hier das ping-Kommando mit der IP-Adresse des Containers `nginx` im Netzwerk `demo_net` aus (Abb. 10.13). Durch die Angabe des Parameters `-c` kann man die Anzahl der `ping`-Versuche begrenzen.

***Lösung:***

```
> docker attach my_alpine
# ping -c 5 172.19.0.2
```

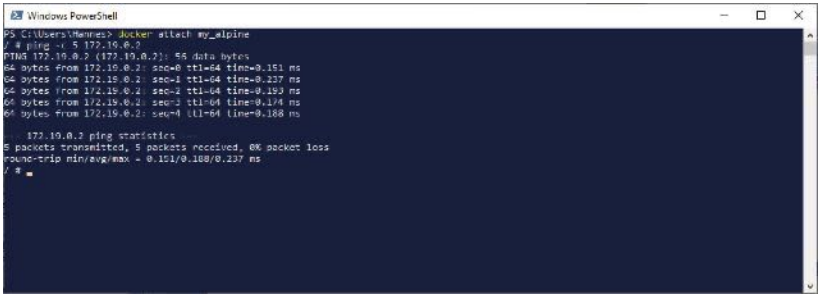



**Abb. 10.13** Zweiter Screenshot zur Netzwerk-Übungsaufgabe

Wie Sie vielleicht wissen, sollte man in der Praxis nie direkt mit IP-Adressen arbeiten, sondern stattdessen einen Host-Namen verwenden. Zum Glück haben Docker Container einen Integrierten DNS Service, der Container-Namen zu deren IP auflöst. So können wir den Container-Namen zur Angabe des Ziel-Hosts verwenden.



Probieren Sie das `ping`-Kommando mit dem Container-Namen aus (Abb. 10.14).

***Lösung:***

```
1  # ping -c 5 nginx
```

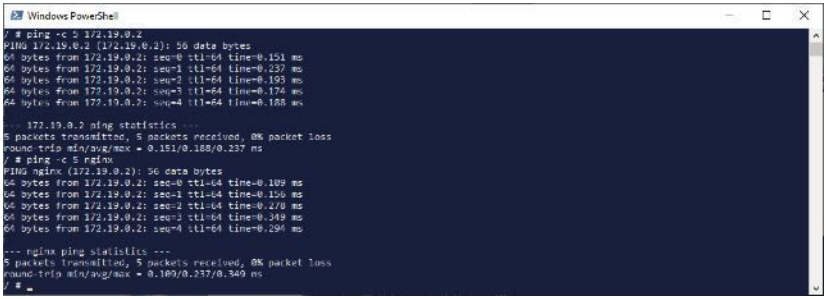



**Abb. 10.14** Dritter Screenshot zur Netzwerk-Übungsaufgabe

Zuletzt versuchen wir noch, aus unserem Container heraus ein ping-Kommando zu einer Web URL abzusetzen. Geben Sie `ping`-Kommando ein, das google.com als Zielangabe verwendet (Abb. 10.15).

***Lösung:***

```
1  # ping -c 5 google.com
```

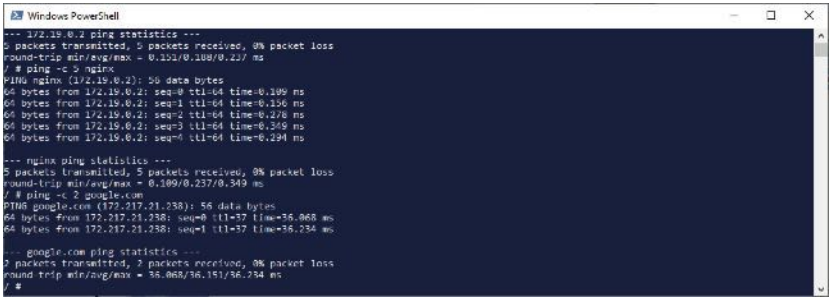



**Abb. 10.15** Vierter Screenshot zur Netzwerk-Übungsaufgabe

Um die Shell des Containers wieder zu verlassen, geben sie das Kommando `exit` ein.

Kapitel 11

# Erstellen eines WordPress-Blogs

Wie bereits im Kapitel „4.4 Häufig verwendete Docker Images“ dieses Buches aufgeführt wurde, ist WordPress im Moment das weltweit am meisten eingesetzte Content-Management-System (CMS), um Webseiten zu erstellen.

WordPress wurde in der Sprache PHP entwickelt und verwaltet seine Daten in einer SQL basierten Datenbank. Dabei kann entweder MySQL oder MariaDB als Datenbank-Service eingesetzt werden.

An dieser Stelle soll jetzt ein Beispiel vorgestellt werden, das eine etwas komplexere Anwendung für eine WordPress-Umgebung mit Docker Containern realisiert. Dazu benötigen wir eine Multi-Container-Umgebung mit zwei Docker Containern. Ein Container kapselt den SQL-Datenbank-Service, der andere verwaltet die WordPress-Applikation, die den Webserver für die Entwicklung von Webseiten zur Verfügung stellt und auch die WordPress-Dateien verwaltet.



## 11.1 Datenbank-Container starten

Für unser Übungsbeispiel erstellen wir als Erstes den Container für die SQL-Datenbank.

Um diesen Datenbank-Container zu starten, laden wir zunächst das benötigte Image von Docker Hub herunter. Hier der Befehl dazu:

```
1  > docker image pull mysql:5.7
```

Derzeit arbeitet WordPress nur mit der Version 5.7 von MySQL korrekt zusammen. Darum müssen wir hier 5.7 explizit als Version angeben. Mit `latest` würde die Verbindung von WordPress zur Datenbank nicht funktionieren.

Damit der Inhalt dieser Datenbank über die Container-Lebenszeit hinaus vorhanden bleibt und nach dem Entfernen des zugehörigen Containers nicht jedesmal neu erstellt werden muss, legen wir diese in einem Volume ab. Dadurch werden, wie in Kap l 8.1 beschrieben, die Daten persistent. Sie können damit nur noch verloren gehen, wenn das Volume explizit gelöscht wird. Wir geben diesem Volume den Namen `"database_data"`.

Hier das Kommando, um dieses Volume zu erstellen:

```
> docker volume create database_data
```

Damit die beiden Container der WordPress-Anwendung später miteinander kommunizieren und Daten austauschen können, benötigen sie ein gemeinsames Docker-Netzwerk. Dieses Netzwerk bekommt in unserem Beispiel den Namen `"wordpress_net"`.

Der Befehl dazu lautet:

```
> docker network create wordpress_net
```

Jetzt haben wir die Voraussetzungen geschaffen, um für unsere WordPress-Anwendung den Datenbank-Container zu starten. Der Befehl dazu ist jetzt schon etwas aufwändiger als bei unseren bisherigen Beispielen:

```
> docker container run -d --name database '
-e MYSQL_ROOT_PASSWORD=mypassword -e MYSQL_DATABASE=wordpress '
-e MYSQL_USER=wordpress -e MYSQL_PASSWORD=wordpress '
--network wordpress_net -v database_data:/var/lib/mysql '
mysql:5.7
```

Zur Wiederholung: Bei der mehrzeiligen Darstellung des Kommandos wurden hier Backticks ( ` ) zum Maskieren des Zeilenendes verwendet. Dies ist die Variante für die Eingabe in einer PowerShell. Bei anderen Shell-Applikationen, wie zum Beispiel der Ubuntu Shell, ist das Zeichen zum Maskieren von Sonderzeichen wie einem Zeilenende in der Regel der Backslash ( \ ).

Falls Sie also nicht mit der PowerShell arbeiten, dann ersetzen Sie das Backtick-Zeichen aus dem obigen Beispiel durch einen Backslash.

Beim Start des Containers haben wir die folgenden Kommandozeilen-Parameter angegeben:

| | |
|---|---|
| `-d` | Der Container wird im detached Mode ausgeführt. |
| `--name` | Der Container-Name ‚database' wird zugewiesen. |
| `-e MYSQL_ROOT_PASSWORD` | Durch die Environment-Variable wird das Root-Passwort des Datenbank-Containers mit ‚mypassword' angegeben. |
| `-e MYSQL_DATABASE` | Durch die Environment-Variable wird als Datenbankname für die SQL-Datenbank ‚wordpress' angegeben. |
| `-e MYSQL_USER` | Durch die Environment-Variable wird als Benutzername für die Anmeldung des WordPress Containers an den Datenbank-Container ‚wordpress' angegeben. |
| `-e MYSQL_PASSWORD` | Durch die Environment-Variable wird als Passwort für die Anmeldung des WordPress Containers an den Datenbank-Container ‚wordpress' angegeben. |
| `--network` | Der SQL Container wird an das Docker-Netzwerk ‚wordpress_net' angebunden. |
| `-v` | Das Volume ‚database_data' wird in den containerinternen Pfad ‚/var/lib'mysql' montiert. |
| `mysql:5.7` | Der Container soll auf Basis des Images ‚mysql' aus dem Docker Hub in der Version 5.7 erstellt werden. |

Wenn als Ergebnis dieses Kommandos ein Digest ausgegeben wird, dann war der Start des Containers erfolgreich. Prüfen wir das sicherheitshalber noch einmal:

```
1 > docker container ls
```

Der folgende Screenshot zeigt den Ablauf bei der Einrichtung und beim Start des Datenbank-Containers (Abb. 11.1):

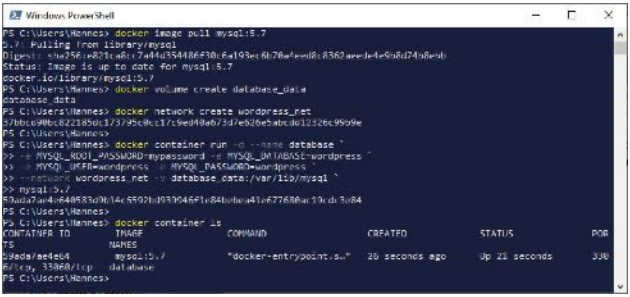



**Abb. 11.1** Manueller Start des SQL-Datenbank-Containers für WordPress

Bevor der WordPress Container gestartet wird, muss sichergestellt sein, dass der Datenbank-Container gestartet und die Datenbank initialisiert ist. Um das zu prüfen, lesen Sie die Log-Informationen des Datenbank-Containers aus.

```
1 > docker container logs database
```

Wird dort der Eintrag `"ready for connection"` angezeigt, dann kann mit dem Start des WorPress Containers begonnen werden (Abb. 11.2).

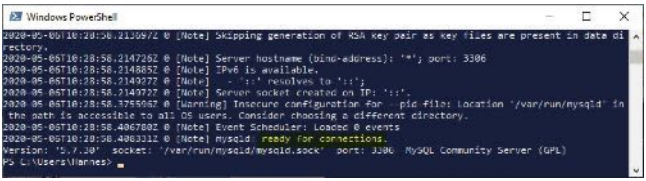



**Abb. 11.2** Der SQL-Datenbank-Container ist `ready for connection`.

## 11.2 WordPress Container starten

Auch hier laden wir als Erstes das WordPress Image vom Docker Hub auf unseren Host-Rechner herunter:

```
> docker image pull wordPress:latest
```

Da wir mit der aktuellsten Version von WordPress arbeiten wollen, geben wir hier als Versionsangabe `latest` ein.

Starten Sie jetzt eine Shell und stellen Sie sicher, dass Sie sich dort in Ihrem Benutzerverzeichnis befinden, das heißt z.B. unter Windows `C:\Users\<username>`.

Mit dem folgenden Kommando starten Sie den Docker Container für WordPress:

```
> docker container run -d --name wordpress '
-e WORDPRESS_DB_HOST=database:3306 -e WORDPRESS_DB_
USER=wordpress '
-e WORDPRESS_DB_PASSWORD=wordpress -e WORDPRESS_DB_
NAME=wordpress '
--link database:database --network wordpress_net '
-v c:\wordpress:/var/www '
-p 8080:80 wordpress:latest
```

Auch hier wird nach erfolgreichem Start des Containers der Digest ausgegeben.

Noch einmal eine Übersicht der verwendeten Parameter mit einer kurzen Beschreibung der Funktion:

| | |
|---|---|
| `-d` | Der Container wird im detached Mode ausgeführt. |
| `--name` | Der Container-Name ‚wordpress' wird zugewiesen. |

| | |
|---|---|
| `-e WORDPRESS_DB_HOST` | Durch die Environment-Variable wird angegeben, dass der WordPress Container auf den SQL-Datenbank-Container mit dem Namen ‚database' über den Port mit der Nummer 3306 zugreifen soll. |
| `-e WORDPRESS_DB_USER` | Durch die Environment-Variable wird als Benutzername für den SQL-Datenbank-User ‚wordpress' angegeben. |
| `-e WORDPRESS_DB_PASSWORD` | Durch die Environment-Variable wird als Passwort für die Anmeldung des WordPress Containers an den Datenbank-Container ‚wordpress' angegeben. |
| `-e WORDPRESS_DB_NAME` | Durch die Environment-Variable wird als Datenbank-Name für die WordPress-Datenbank ‚wordpress' angegeben. |
| `--link database:database` | Stellt eine Verbindung vom WordPress Container zum Container ‚database' her. |
| `--network` | Der WordPress Container wird an das Docker-Netzwerk ‚wordpress_net' angebunden. |
| `-v` | Das lokale Host-Verzeichnis ‚c:\wordpress' wird in den containerinternen Pfad ‚/var/www' montiert. |
| `-p` | Der interne Port 80 wird nach außen auf Port 8080 gemappt. |
| `wordpress:latest` | Der Container soll auf Basis des Images ‚worpress' aus dem Docker Hub in der aktuellsten Version erstellt werden. |

Wir prüfen jetzt, ob die beiden Container laufen, die wir für unsere WordPress-Anwendung benötigen:

```
1  > docker container ls
```

Hier der Screenshot einer PowerShell beim Start des WordPress Containers (Abb. 11.3):

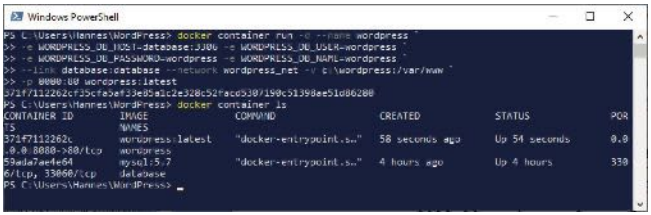



**Abb. 11.3** Manueller Start des WordPress Containers

Ab sofort können Sie auf Ihre WordPress-Applikation über einen Webbrowser Ihrer Wahl zugreifen.

Geben Sie in die Adresszeile folgende Adresse ein:

http://localhost:8080/wp-admin/install.php

Wenn alles richtig installiert ist, dann erscheint die erste Seite der WordPress-Installationsroutine.

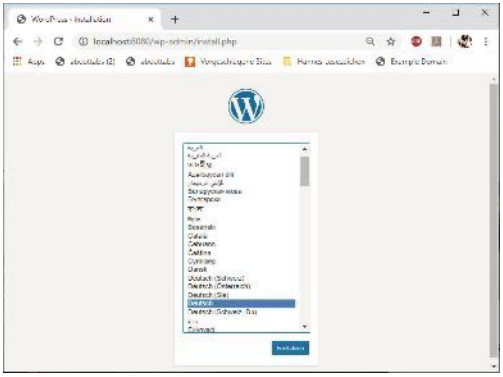


**Abb. 11.4** WordPress-Startseite zur Sprachauswahl

Wählen Sie auf dieser Seite die gewünschte Sprache aus (hier Deutsch) und klicken Sie auf den Button [Fortfahren] (je nach Sprache zum Beispiel [Continue] bei Englisch als gewählte Sprache).

Als nächster Schritt folgt die Willkommensseite der WordPress-Installationsroutine.

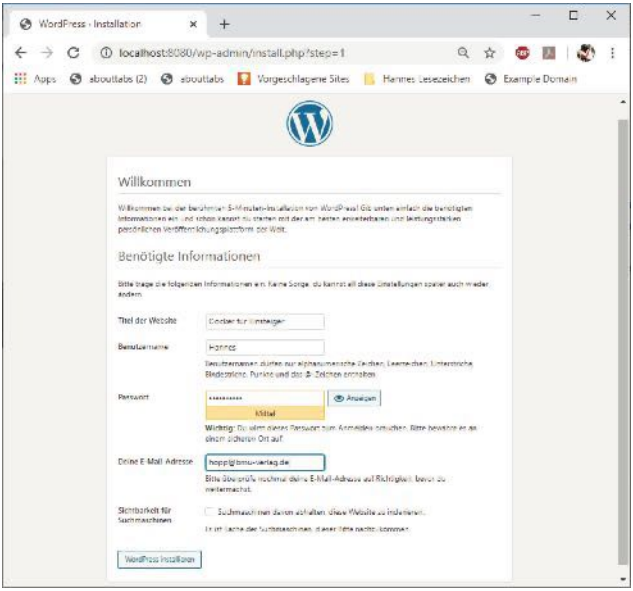



**Abb. 11.5** Die WordPress-Willkommensseite

Tragen Sie auf dieser Seite den Titel der Webseite, den von Ihnen gewählten Benutzernamen und ein sicheres Passwort ein. Zuletzt geben Sie eine E-Mail-Adresse an und starten die Installation von WordPress in Ihrem Container durch einen Klick auf die Schaltfläche [WordPress installieren].

Nach erfolgreicher Installation wird die Installationsseite #2 eingeblendet.

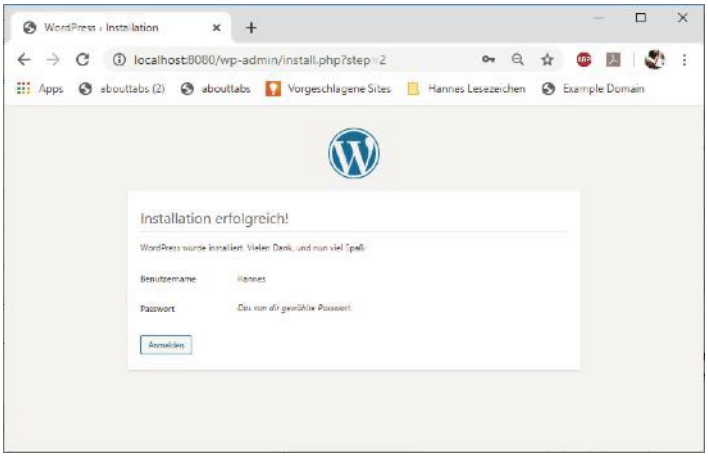



**Abb. 11.6** Die WordPress-Installationseite #2

Noch ein Mausklick auf den Button [Anmelden] und es geht weiter zur eigentlichen Login-Seite, wo Sie den gerade angegebenen Usernamen und Ihr Passwort eingeben. Schließlich klicken Sie noch einmal auf den Button [Anmelden], um den Login-Vorgang abzuschließen.



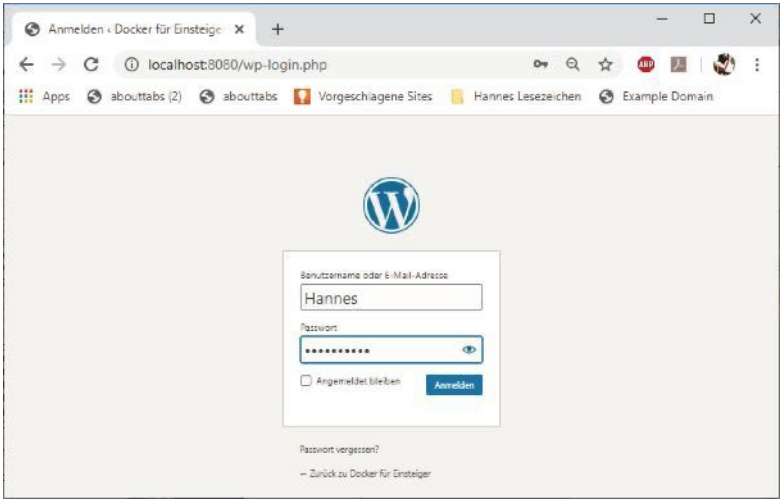



**Abb. 11.7** Die WordPress-Login-Seite

Nun haben Sie es geschafft. Sie sind im WordPress-Dashboard und können damit beginnen, Ihre Homepage zu gestalten.

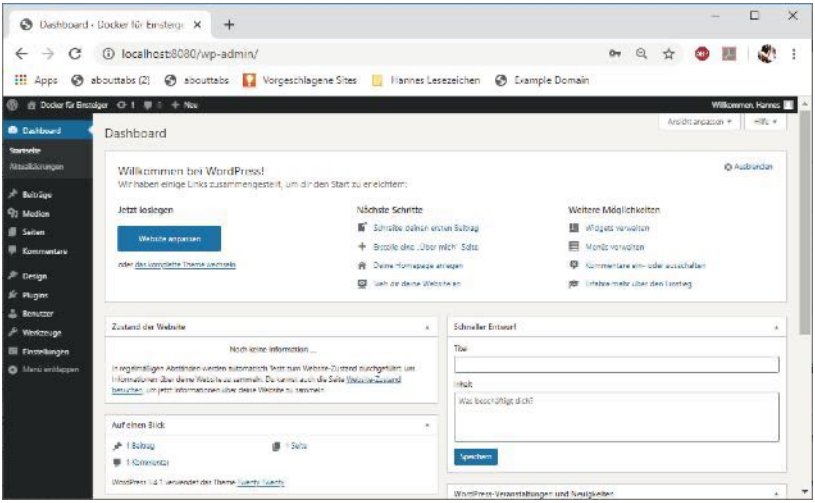



**Abb. 11.8** Die WordPress-Seite mit dem Dashboard

Wenn Sie sich über die Erstellung und Bearbeitung von Webseiten und Blogs mit WordPress umfassender informieren wollen, empfehlen ich Ihnen das Buch „WordPress 5 für Einsteiger“ vom BMU Verlag.

Hier ein Link für die Webseite des BMU Verlages:

https://bmu-verlag.de/wordpress/

## 11.3 Aufräumen der WordPress-Anwendung

Da wir später unsere WordPress-Anwendung noch mit Docker Compose erstellen wollen, räumen wir erst noch einmal auf:

Stoppen Sie zuerst die beiden Container der WordPress-Anwendung:

```
1 > docker container stop database wordpress
```

Wir fahren fort und entfernen als Nächstes die beiden Container:

```
> docker container rm database wordpress
```

Die Images brauchen wir im Moment auch nicht mehr:

```
> docker image rm mysql:5.7 wordpress
```

Wir entfernen das Volume für unseren Datenbank-Container:

```
> docker volume rm database_data
```

Zuletzt räumen wir das Docker-Netzwerk auf:

```
> docker network rm wordpress_net
```

In diesem Kapitel wurde Ihnen gezeigt, wie Sie die benötigten Docker Container für eine WordPress-Anwendung manuell starten und vernetzen können.

Im folgenden Kapitel lernen Sie den Umgang mit Docker Compose. Danach zeigen wir Ihnen, wie man eine WordPress-Applikation mithilfe von Docker Compose recht komfortabel aufsetzen, starten und stoppen kann.

Kapitel 12

# Docker Compose

## 12.1 Was ist Docker Compose

In Kapitel 7 haben Sie zu den bewährten Praktiken bei der Arbeit mit Docker unter anderem erfahren, dass jeder Container jeweils nur eine einzige, wohldefinierte Aufgabe erfüllen soll. Das bedeutet, dass jeder Prozess in einem eigenen Container laufen sollte. Natürlich könnten wir jetzt jeden Docker Container einzeln über Shell-Kommandos starten, per Hand die benötigen Volumes einbinden und auch die Netzwerke zuordnen, die zur Kommunikation der Container untereinander nötig sind.

Bei aufwendigen Diensten mit zahlreichen Containern kann das allerdings sehr schnell unübersichtlich werden. Es eröffnen sich dann viele Möglichkeiten für Fehler, die in vielen Fällen nur mit sehr viel Zeitaufwand gefunden und behoben werden können.



Jetzt wird sich die eine oder andere Leserin, beziehungsweise der eine oder andere Leser überlegen, dass man dafür ein Shell-Script erstellen könnte. Diejenigen Leser, die eine Programmiersprache wie Python beherrschen, kommen vielleicht auf die Idee, ein Programm zu entwickeln, das die anfallenden Aufgaben erledigt und die Informationen dafür mit einer Konfigurationsdatei übergeben bekommt.

Sie können sich diese Arbeit ersparen. Das Python-Programm gibt es schon – Docker Compose.

Um die Konfiguration und die Kontrolle komplexer Systeme zu erleichtern, stellt uns Docker das Tool Docker Compose zur Verfügung. Damit wird es möglich, komplexe Lösungen mit mehreren Containern zu definieren, zu verwalten, zu starten und auch wieder anzuhalten.

Die folgenden Merkmale kennzeichnen die Vorzüge von Docker Compose:

- Mehrere entkoppelte Laufzeitumgebungen können auf dem gleichen Host-Rechner ausgeführt werden. Es kann über Docker Compose ein Projektname vergeben werden, um die Laufzeitumgebungen zu entkoppeln.
- Erhaltung von Volume-Daten, welche von Services erstellt beziehungsweise benötigt werden. Werden Container erneut gestartet, stehen alle Daten aus vorherigen Container-Versionen immer noch bereit.
- Docker-Compose bietet die inkrementelle Aktualisierung der Container für Services an. Nur Container, welche Änderungen aufweisen, werden vor einem Start neu gebaut. Änderungen an einem Service nehmen dadurch sehr wenig Zeit in Anspruch.
- Compose erlaubt es, Umgebungsvariablen in Compose-Dateien zu benutzen. Mit diesen Variablen kann man beispielsweise eine Komposition an verschiedene Umgebungsbedingungen und Benutzer anpassen.

Docker Compose kann Container nur auf dem lokalen Host starten. Docker Compose eignet sich also nicht für Lösungen zur Hochverfügbarkeit von Diensten und kann auch nicht das Load-Balancing verwalten, also die Lastenverteilung von Diensten. Für diese Aufgaben kommen Orchestrierungs-Anwendungen wie Docker Swarm oder Kubernetes zum Einsatz.

Die Konfiguration von Applikationen erfolgt über eine Konfigurationsdatei, die im YAML-Format erstellt wird. Mit einem einzigen Kommando werden alle benötigten Container-Dienste aus der Konfigurationsdatei heraus gestartet.

## 12.2 Installation von Docker Compose

Da Docker für seine Arbeit die Docker Engine unbedingt benötigt, muss diese als Voraussetzung installiert worden sein. Falls Sie Docker Desktop unter Windows oder unter MAC-OS installiert haben, dann beinhaltet diese Installation bereits Docker Compose.

Um zu festzustellen, ob Docker Compose bei Ihnen installiert und aktiv ist, öffnen Sie eine Kommando-Shell und geben das folgende Kommando ein (Abb. 12.1):

```
> docker-compose version
```

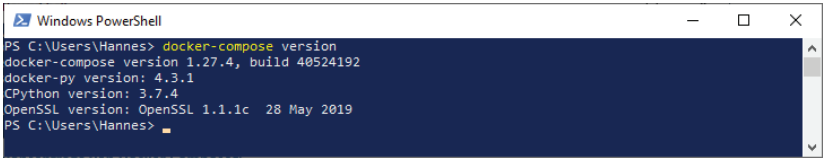



**Abb. 12.1** Versionsabfrage von Docker Compose

Erhalten Sie, wie im Screenshot vorgeführt, die Versionsinformationen, dann können Sie davon ausgehen, dass Docker Compose korrekt installiert wurde.



### 12.2.1 Installation unter Linux

Um Docker Compose auf einem Linux-System zu installieren, kann man die Binärdateien von der „Compose Repository Release Page“ aus dem GitHub herunterladen.

Führen Sie dazu die folgenden Schritte aus (das `curl`-Kommando muss für diese Anleitung auf Ihrem System installiert sein).

1. Starten Sie den Download der aktuellen Release-Version von Docker Compose:

```
sudo curl -L \
```

```
"https://github.com/docker/compose/releases/download/<VERSION>/
docker-compose-$(uname -s)-$(uname -m)" \
-o /usr/local/bin/docker-compose
Setzen Sie für <VERSION> die von Ihnen gewünschte Versionsnummer
für Docker Compose ein (z.B. 1.25.4)
```

2. Aktivieren Sie das Execution-Recht für die ausführbare Datei:

```
sudo chmod +x /usr/local/bin/docker-compose
```

3. Falls die Ausführung des Kommandos `docker-compose` fehlschlägt, dann ergänzen Sie die Pfad-Angabe im `path`-Eintrag für Ihr System oder erstellen Sie alternativ mit dem folgenden Linux-Kommando einen symbolischen Link im Verzeichnis `/usr/bin`.

```
sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
```

Prüfen Sie auch hier die Installation durch die Versionsabfrage von Docker Compose:

```
> docker-compose version
```

Der folgende Screenshot zeigt die Installation von Docker Compose mit den oben vorgestellten Kommandos über eine Ubuntu Shell (Abb. 12.2).

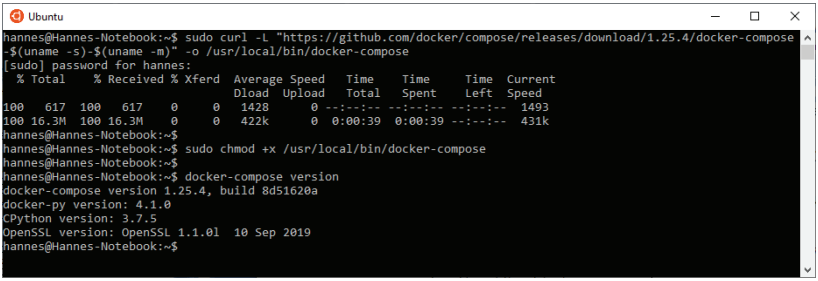



**Abb. 12.2** Installation von Docker Compose unter Ubuntu Linux

## 12.3 Das YAML-Format

Docker Compose benutzt eine Textdatei, mit deren Hilfe man die Dienste einer Applikation konfiguriert. Diese Datei muss im YAML-Format erstellt werden und muss unter dem Namen `'dockercompose.yml'`oder ‚`dockercompose.yaml'` abgespeichert werden.

Bei YAML handelt es sich um eine einfache Auszeichnungssprache, die von XML abgeleitet wurde. YAML war ursprünglich die Abkürzung für *Yet Another Markup Language* und wurde später in ein sogenanntes Retronym umdefiniert. Es bedeutet jetzt *YAML Ain't Markup Language*. Damit soll deutlich gemacht werden, dass das Format datenorientiert ist und nicht als Auszeichnungssprache für Dokumente gedacht ist.

Wenn Sie bereits Erfahrung mit dem JSON-Format haben, dann wird Ihnen wahrscheinlich bei YAML-Dateien einiges bekannt vorkommen. JSON ist eine Untermenge von YAML und JSON-Dateien sind ab Version 1.2 von YAML gültige YAML-Dateien.

Docker-Compose-Dateien spezifizieren die Eigenschaften von Services, Netzwerken und Volumes für Docker Applikationen. Beim Aufruf des Kommandos `docker-compose` wird standardmäßig nach der folgenden Datei gesucht:



```
1 './docker-compose.yml' bzw. ,./docker-compose.yaml'
```

### 12.3.1 YAML-Elemente in Compose-Dateien

Wir stellen Ihnen hier kurz die wichtigsten Elemente von YAML-Dateien vor. Dabei liegt der Schwerpunkt auf Elementen, die man für Entwicklung von Docker Compose YAML-Dateien benötigt.

Das Hash-Zeichen leitet einen einzeiligen Kommentar ein.

```
1 # Das ist ein Kommentar
```

Basis-Elemente in Compose-Dateien sind Listen und Mappings. Diese Elemente können untereinander verschachtelt sein und so komplexere Objekte bilden.

Listen werden auch Sequenzen genannt. Jedes Element steht in einer eigenen Zeile und beginnt mit einem Bindestrich.

```
- peter
- rudy
- hans
```

Sequenzen können beliebige YAML-Daten enthalten, auch Sub-Sequenzen. Eine Sequenz innerhalb einer anderen Sequenz beginnt mit einem Bindestrich ohne Wert, danach folgen die Elemente der Sub-Sequenz als eingerückte Liste.

```
- entwicklung
-
    - peter
    - rudy
    - hans
- test
-
    - helga
    - klara

```

Mappings: Sie werden auch als Dictionaries bezeichnet, sind Key-Value-Paare, die durch einen Doppelpunkt getrennt werden.

```
Version: 3.7
```

Ein Mapping kann als Wert auch eine Liste enthalten.

```
expose:
    - "3000"
    - "8000"
```

Der Wert eines Mappings kann wiederum aus weiteren, verschachtelten Mappings zusammengesetzt sein.

```
build:
  context: .
  labels:
    com.example.description: "Accounting webapp"
    com.example.department: "Finance"
    com.example.label-with-empty-value: ""
```

### 12.3.2 Sektionen in Docker Compose YAML-Dateien

Dieser Abschnitt stellt Ihnen eine Auswahl der gebräuchlichsten YAML-Objekte vor, die in Docker Compose Konfigurationsdateien als Sektionsangaben verwendet werden.

| | |
|---|---|
| `version:`<br><br>`version: „3.7“` | Angabe der benutzten Version für das Compose Datei-Format. |
| `services:` | Mit diesem Element beginnt der Abschnitt, in dem die Informationen über die zu startenden Container eingetragen werden. |
| `networks:` | Unter dieser Sektion der ersten Ebene definiert man die Netzwerke, die durch Docker Compose erstellt werden sollen. |
| `volumes:` | Obwohl es möglich ist, Volumes als Bestandteil der Service-Deklarationen zu spezifizieren, ermöglicht diese Sektion Volumes zu erstellen, die über alle Services hinweg verwendet werden können. |

#### *12.3.2.1 Sektion Services*

In dieser Sektion definieren wir zunächst die Namen der Services und unter den Namenselementen werden die Eigenschaften der Services angegeben.

```
services:
    web:
        …

    database:
        …
```

Unterhalb des Service-Namens folgen weitere Angaben zu den Eigenschaften eines Service.

| | |
|---|---|
| `image:` | Die Image ID eines lokal oder entfernt liegenden Images. Docker Compose versucht das Image zu pullen, wenn es lokal nicht gefunden wird. |

```
services:
    web:
        image: nginx:1.17.7
        …

```

| | |
|---|---|
| `build:` | Mit diesem Schlüsselwort definieren Sie die Angaben, die benötigt werden, um einen Container aus einem Image zu erstellen, welches aus einem eigenen Dockerfile gebaut werden soll.<br><br>Es stehen für die YAML-Dateien zwei Varianten zur Verfügung.<br><br>Zum einen kann man hier als Wert nur den Pfad zu dem Verzeichnis eintragen, das den Build-Context beinhaltet. Die Pfadangabe soll relativ zu dem Verzeichnis sein, in dem sich die Docker-Compose-Datei befindet.<br><br>Zum anderen kann das Schlüsselwort `build` eine eigene Sektion einleiten. Damit ist es möglich, zusätzliche Parameter anzugeben. Als Schlüsselwörter stellen wir hier `context`, `dockerfile` und `args` vor. |
| `context:` | Der Pfad zum Dockerfile-Verzeichnis oder die URL zu einem Git Repository. |

| | |
|---|---|
| `dockerfile:` | Angabe eines alternativen Dockerfiles, um das Image zu bauen. |
| `args:` | Hier können Sie zusätzliche `build`-Argumente eingetragen, welche wiederum Environment-Variablen repräsentieren, die Docker im Laufe des Build-Prozesses auswertet. Diese müssen im Dockerfile mit dem Schlüsselwort ARG spezifiziert worden sein. |
| `stdin_open: true` | Dieser Eintrag entspricht dem Parameter `-i` beim `docker run`-Kommando. Setzen Sie diesen Wert auf `true`, um den interaktiven Modus zu aktivieren. |
| `tty: true` | Diese Angabe entspricht dem Parameter `-t` des `docker run`-Kommandos. Hat dieser Eintrag den Wert `true`, wird ein TTY aktiviert. |

Beispiel für die erste Variante:

```
services:
    web:
        build: ../hello-web

        …
```

Beispiel für die zweite Variante:

```
services:
    web:
        build:
            context: ../hello-web
            dockerfile: my_dockerfile
            args:
                relno: 1.0.0
        …

```

| | |
|---|---|
| `ports:` | Für jeden Service kann mit diesem Element angegeben werden, welche Ports veröffentlicht werden. |

```
services:
    web:
        image: nginx:1.17.7
        ports:
            - "4000"
            - "8080:80"
            …
        …

```

| | |
|---|---|
| `networks:` | Für jeden Service kann mit diesem Element angegeben werden, in welche Netzwerke dieser eingebunden werden soll. Die Namen dieser Netzwerke werden in der Top-Level-Sektion `networks` festgelegt. |

```
services:
    web:
        image: nginx:1.17.7
        …
        networks:
            my_net:
            …
        …

```

| | |
|---|---|
| `volumes:` | Hier werden die Pfadangaben für Volumes angegeben, die in den zugehörigen Service eingebunden werden sollen. |

```
services:
    web:
        image: nginx:1.17.7
        …
        volumes:
            - ./HTML:/usr/share/nginx/html
```

```
            …
        …
```

| | |
|---|---|
| `environment:` | In diesem Abschnitt kann man einem Service Environment-Variablen zufügen. |

```
services:
    web:
        image: nginx:1.17.7
        …
        environment:
            DEBUG: 'true'
            CONFIG_FILE_PATH: /code/config
            …
        …
```

#### *12.3.2.2. Networks*

Wie schon in der Übersicht erwähnt, definiert man hier die Netzwerke, die durch Docker Compose erstellt werden sollen und die von den Containern der Anwendung zur Kommunikation verwendet werden können. Diese werden in der Sektion `networks` der entsprechenden Services eingetragen.



```
networks:
    my_net:
    …
```

#### *12.3.2.3 Volumes*

In dieser Sektion befinden sich die Angaben zu Volumes, welche von allen Services eingebunden und zum Zugriff auf gemeinsame Daten verwendet werden können.

```
volumes:
    my_database:
    …
```

## 12.4 Ein erstes Docker Compose YAML-Beispiel

Wir beginnen mit einer ersten einfachen Übung und erstellen wieder einen NGINX Container wie aus dem Kapitel 4.8, diesmal aber mithilfe von Docker Compose.

Zunächst legen Sie wieder ein neues Verzeichnis mit dem Namen „`Hello-Compose`“ unter Ihrem Benutzerverzeichnis an.

Erzeugen Sie in diesem Verzeichnis eine Textdatei mit dem Dateinamen `'docker-compose.yaml'`.

Fügen Sie in diese Datei die folgenden Einträge ein:

```
1  Datei ,docker-compose.yaml'
2  version: "3.7"
3  services:
4      my_web:
5          image: nginx:1.17.7
6          ports:
7              - "8080:80"
```

In der ersten Zeile geben wir die Version der verwendeten Docker-Compose-Syntax an.

In der zweiten Zeile steht das Startelement für die Sektion `services`.

In dieser Konfiguration gibt es nur einen Container mit dem Namen `my_web`. In der dritten Zeile steht der Name dieses Containers als Startelement für diese Sektion.

In der vierten Zeile spezifizieren wir den Namen und die Version des Images, welches für den Bau des Containers verwendet werden soll.

Zeile fünf und sechs geben an, dass der interne Port 80 auf dem externen Port 8080 veröffentlicht werden soll. Das entspricht in diesem Fall dem Parameter `-p 8080:80`, den wir schon beim `docker run`-Kommando angegeben haben.

## 12.5 Up and Down

Wir starten den Container jetzt nicht mehr mit dem Kommando `docker run`, sondern wir nutzen die Applikation *Docker Compose*. Starten Sie dazu eine Kommando-Shell und wechseln Sie in das Verzeichnis, in dem sich die Datei `'docker-compose.yaml'` befindet.

Dort geben Sie den folgenden Befehl ein:

```
1 > Docker-compose up -d
```

Dieses Kommando startet alle Container, die in der Compose-Datei definiert sind. Die Compose-Datei muss sich im aktuellen Verzeichnis befinden. Der Parameter `-d` (detached) im obigen Beispiel bewirkt, wie beim Kommando `docker run` auch, dass die Container im Hintergrund gestartet werden und man in der aktuellen Kommando-Shell wieder zur Eingabezeile zurückkommt.

Sehen wir doch noch kurz nach, ob der Container `nginx` wirklich aktiv ist. Das prüfen wir mit diesem bereits bekannten Kommando:

```
1 > docker container ls
```



Jetzt können Sie sich in einem Internet-Browser die Webseite aus dem Image von NGINX anzeigen lassen. In der Adresszeile des Browsers geben Sie die bekannte URI http://localhost:8080/ ein.

Gestoppt werden alle Container eines Dienstes mit dem Parameter `down` des `docker-compose`-Kommandos.

```
1 > docker-compose down
```

Noch einmal prüfen wir mit Befehl `docker container ls`, ob der Container tatsächlich beendet wurde. Wenn Sie die Anzeige der Webseite im Browser aktualisieren, dann werden Sie nur noch darüber informiert, dass die Seite nicht erreichbar ist.

Hier noch ein Screenshot eines Kommandofensters, in dem die gerade angesprochenen Kommandos ausgeführt worden sind (Abb. 12.3):


**Abb. 12.3** Die Kommandos `docker-compose up` und `docker-compose down`

## 12.6 Das NGINX-Beispiel erweitern

Natürlich wollen wir jetzt wieder, dass unsere eigene Webseite aufgerufen wird, zum Beispiel die aus dem Kapitel 4.8.2 dieses Buches.

Wir erzeugen dazu unter dem Verzeichnis mit der Docker-Compose-Datei (`<USER_DIR>/Hello Compose`) ein neues Verzeichnis mit dem Namen `html`. Dort legen wir die Datei `index.html` an, welche die Einträge für unsere eigene Webseite enthält.

Hier die Variante aus dem Kapitel 4.8.2 mit ein paar Veränderungen:

```
Datei ,index.html'
<!DOCTYPE html>
<html lang="en">
<head>
      <meta http-equiv="content-type" content="text/html;
      charset=utf-8"/>
      <meta name="description" content="Eine einfache Webseite
      für Nginx"/>
      <meta name="author" content="Hans-M. Hopp"/>
      <meta name="keywords" content="Docker, Handbuch, Nginx,
      Hello, Hello Web"/>
      <meta name="date" content="2019-12-05"/>
      <!-- <script type="text/javascript" charset="utf-8"
      src="file.js"></script> -->
      <link rel="stylesheet" href="styles.css" type="text/css"
      />

  <!-- Title
```

```
  ------------------------------------------------- -->
  <title> Hello Compose </title>

</head>
<body>

  <div class="container">
    <div class="row" style=”margin-top: 10%; margin-left: 10%">
          <h1 style="color:red">Hello Compose!</h1> <br/>
          <p>Diese Seite wird in einem <strong>docker</strong>
<br/>
          container mit Nginx ausgeführt.</p>
    </div>
      <div class="row" style="margin-top: 5%; margin-left: 10%">
          <h2 style="color:blue">Gestartet mit Docker Compose
          </h2>
</div>
</div>
</body>
</html>
```

Um diese HTML-Datei in unseren Container zu übernehmen, müssen wir nur noch die Compose-Datei ‚`docker-compose.yaml`‘ um die Sektion `volumes` ergänzen.



Die mount-Angabe darunter muss so aufgebaut sein, dass das Verzeichnis mit unserer HTML-Datei in das von NGINX benutzte Standardverzeichnis für HTML-Dateien eingebunden wird.

```
Datei ‚docker-compose.yaml'
version: "3.7"
services:
    my_web:
        image: nginx:1.17.7
        ports:
            - "8080:80"
        volumes:
            - ./HTML:/usr/share/nginx/html
```

Dann wollen wir die Änderungen nur noch testen und starten unseren kleinen Service wieder:

```
> Docker-compose up -d
```

Im Browser die Anzeige unbedingt noch einmal aktualisieren (z.B. durch die Taste [F5] oder den Aktualisierungs-Button Browsers).

Jetzt sollte unsere eigene Webseite angezeigt werden (Abb. 12.4).

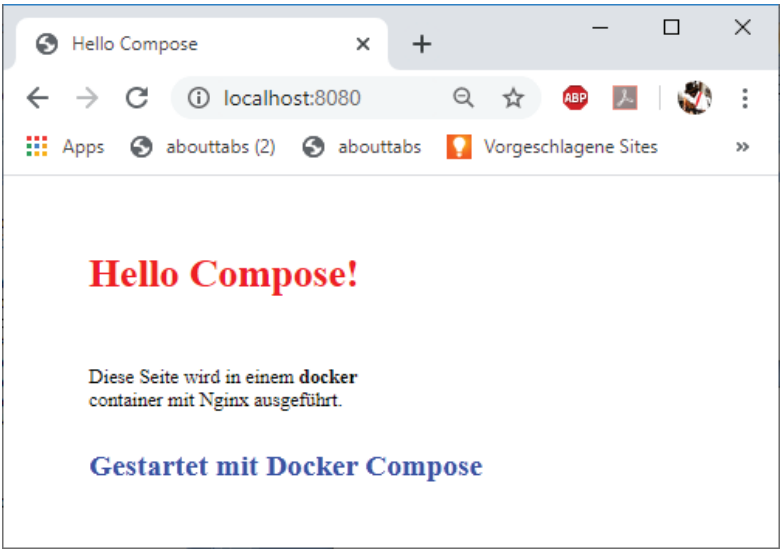



**Abb. 12.4** Eine eigene Webseite mit NGINX und Docker Compose

Bevor wir mit einer neuen Übungsaufgabe beginnen, wollen wir unseren Service noch ordentlich beenden:

```
1 > Docker-compose down
```

Das war schon alles. Alle von Docker Compose gestarteten Container werden mit diesem Kommando gestoppt und entfernt.

## 12.7 Übungsaufgabe: Docker Compose mit eigenem Image

Ihre Aufgabe in dieser Übung ist es, einen Container für die Telefon-PHP-Applikation aus Kapitel 4.9 mit Docker Compose zu erstellen und zu starten.

Hinweise:

Erstellen Sie für die Aufgabe ein neues Verzeichnis. In der Lösung wird der Verzeichnisname „Telefon-Compose“ verwendet.

Da Sie hier einen Container aus Ihrem eigenen Image erstellen wollen, muss in der Docker-Compose-Datei in der Sektion `services` der Abschnitt `image:` durch den Abschnitt `build:` ersetzt werden. Der bekommt folgenden Inhalt:

```
1         build: ../Telefon-PHP
```

***Lösung:***

Die Docker-Compose-Datei:

```
Datei 'docker-compose.yaml'
version: "3.7"
services:

    telefon_app:
        build: ../Telefon-PHP
        ports:
            - "8808:80"
        volumes:
            - ../Telefon-PHP/src/:/var/www/html/
```

Start von Docker Compose aus dem Verzeichnis `<USER_DIR>\Telefon-Compose`. (Abb. 12.5)

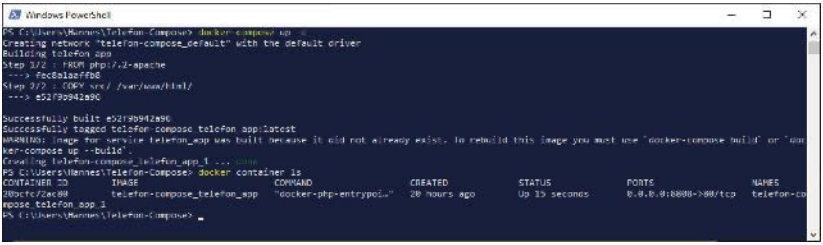



**Abb. 12.5** Docker Compose Übung ‚Telefon-App': Das Kommando `docker-compose up`.

Hier unsere Webseite „Telefon Liste PHP" (Abb. 12.6).

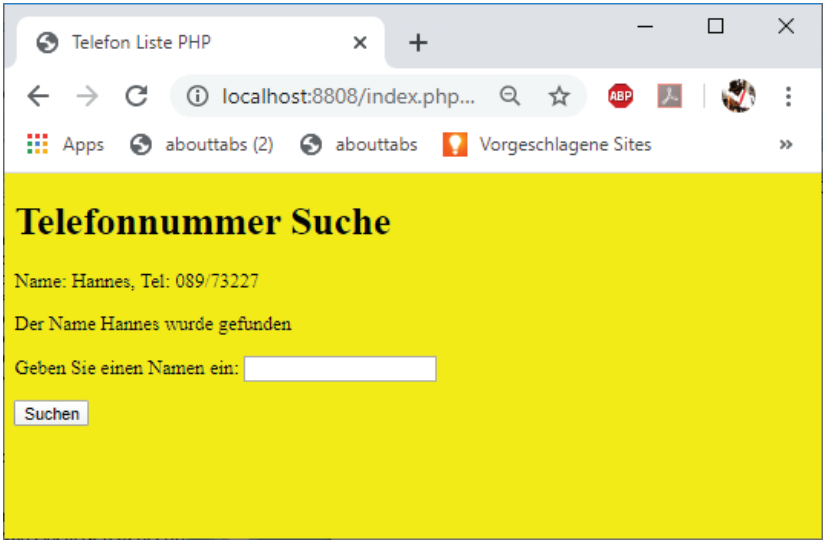



**Abb. 12.6** Docker Compose Übung ‚Telefon-App': die Webseite „Telefon Liste PHP"



## 12.8 Docker Compose mit zwei vernetzten Containern

Unsere ersten Beispiele dienten nur der Einführung in Docker Compose und als praktische Übungsbeispiele. Der Vorteil von Docker Compose kommt aber erst zur Geltung, wenn mehrere Container zusammen die Funktionalität einer Applikation realisieren, wie es bei einer modernen Architektur, die aus mehreren Microservices besteht, der Fall sein sollte.

In unserem nächsten Beispiel erstellen und vernetzen wir zwei verschiedene Container und testen damit die Verbindung zwischen diesen über ein Docker-Netzwerk.

### 12.8.1 Ein Container mit erweitertem Ubuntu Image

Damit wir die Kommunikation später testen können, benötigen wir einen Container, der sowohl das `ping`-Kommando als auch das `curl`-Kommando enthält.

Das Ubuntu Image aus dem Docker Hub enthält diese beiden Applikationen allerdings nicht. Aber wir können dieses Image ja jederzeit erweitern und unser eigenes Image davon ableiten. Dieses wird dabei so erweitert, dass sowohl das `ping`-Kommando als auch das `curl`-Kommando beim Bau des neuen Images installiert wird.

Für dieses Image erstellen wir einen neuen Kontext. Dazu legen wir ein neues Verzeichnis unter unserem Benutzerverzeichnis mit dem Verzeichnisnamen „`UbuntuExt`“ an.

Dort erstellen wir ein neues Dockerfile mit folgendem Inhalt:

```
1  Datei ‚Dockerfile'
2  FROM ubuntu:latest
3  RUN apt-get update && \
4      apt-get install -y curl && \
5      apt-get install -y iputils-ping
6
7  CMD ["/bin/bash"]
```

Wir leiten damit unser eigenes Image von der neuesten Version des Ubuntu Images ab.

Mit dem RUN-Eintrag sorgen wir dafür, dass die Paketlisten von Ubuntu zunächst auf den neuesten Stand gebracht werden. Anschließend wird `curl` installiert und zuletzt erfolgt die Installation des `ping`-Kommandos.

Dies alles wird mit einer einzigen RUN-Anweisung durchgeführt, um die Anzahl der temporären Images im Cache zu reduzieren (siehe Kapitel 7.1.2).

Der CMD-Eintrag, der als letzte Anweisung im Dockerfile stehen sollte, sorgt dafür, dass eine `bash` Shell gestartet wird.

Nachdem wir das Image gespeichert haben, starten wir eine Shell, wechseln in das Verzeichnis `<USER_DIR>/UbuntuExt` und geben das Kommando ein, mit dem unser neues Image gebaut wird (das neue Image soll den Namen `ubuntu-ext` erhalten):

```
1  > docker build -t <DOCKER_ID>/ubuntu-ext .
```

Um das neue Image zu testen, starten wir einen Container mit diesem Image:

```
1  > docker run -it --rm <DOCKER_ID>/ubuntu-ext
```

Der Container wird durch dieses Kommando interaktiv mit einem (Pseudo-)TTY gestartet. Mit dem Parameter `--rm` geben wir an, dass der Container nach dem Beenden automatisch wieder gelöscht wird.

Nachdem Sie den Container gestartet haben, wird Ihnen der Kommando Prompt der `bash` Shell angezeigt.



Zum Testen geben wir in der `bash` Shell des gestarteten Containers ein `ping`-Kommando mit der Angabe des `localhost` als Zieladresse ein. Mit der Angabe `-c 3` legen wir fest, dass das `ping`-Signal dreimal gesendet wird.

```
1  # ping localhost -c 3
```

Anschließend testen wir, ob das `curl`-Kommando korrekt installiert ist. Dazu fragen wir einfach einmal die Version der installierten Anwendung ab.

```
1  # curl -V
```

Der nächste Screenshot zeigt die Ausführung dieser Aktionen in der PowerShell (Abb. 12.7).

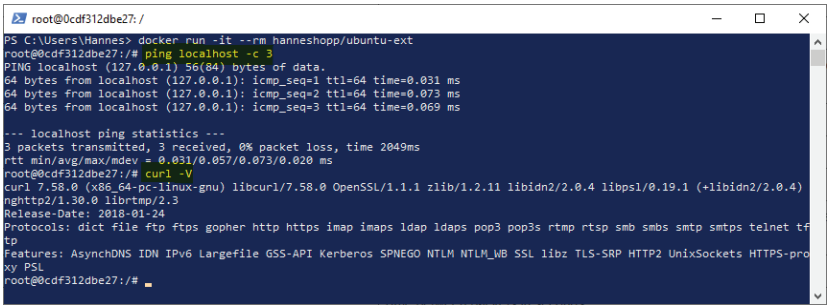



**Abb. 12.7** Test des erweiterten Ubuntu Containers mit `ping` und `curl`

Wenn Sie mit `curl` eine „echte“ URL testen wollen, dann können Sie zum Beispiel auch das folgende Kommando eingeben:

```
1  # curl http://example.com
```

Sie haben richtig gelesen. Diese URL gibt es wirklich. Es handelt sich hier um die „amtliche“ Beispieldomäne, die von der IETF (International Engineering Task Force) als Quasi-Standard bereitgestellt wird. Dieser Domänen-Name ist permanent reserviert.

Selbstverständlich können Sie hier die Applikation `curl` auch mit anderen URLs, wie zum Beispiel `http//www.google.com`, testen.

Verlassen können Sie die `bash` Shell wieder mit dem Kommando `exit`.

### 12.8.2 Erweiterten Ubuntu Container über Docker Compose ausführen

Unser Docker-Compose-Übungsbeispiel wird in einem neuen Verzeichnis aufgebaut. Erstellen Sie dazu das Verzeichnis „`DoubleService`“ unter Ihrem Benutzerverzeichnis.

```
1  <USER_DIR>\DoubleService
```

Legen Sie jetzt in diesem Verzeichnis eine neue Docker-Compose-Datei an und füllen Sie diese mit den folgenden Einträgen:

```
Datei ,docker-compose.yaml'
version: "3.7"
services:

    ubuntu_ext:
        build: ../UbuntuExt
        stdin_open: true
        tty: true
```

Der zusätzliche neue Eintrag `stdin_open: true` erfüllt die gleiche Funktion wie Parameter `-i` beim `docker run`-Kommando. Es wird dadurch der interaktive Modus ermöglicht. Die letzte Angabe `tty: true` entspricht dem Parameter `-t` des `docker run`-Kommandos und aktiviert ein TTY.

Speichern Sie die Docker-Compose-Datei ab und starten Sie eine Kommando-Shell. Wechseln Sie in der Shell in das Verzeichnis `<USER_DIR>\DoubleService` und geben Sie das Kommando zum Start von Docker Compose ein. Der Parameter `-d` sorgt wieder dafür, dass der Container im Hintergrund (detached) ausgeführt wird.



```
> Docker-compose up -d
```

Wir geben das Kommando ein, um uns die Informationen über die laufenden Container anzeigen zu lassen. Damit finden wir den Namen des Containers heraus.

```
> docker container ls
```

Der neue Container hat den Namen `doubleservice_ubuntu_ext_1`.

Wie Sie erkennen können, bildet Docker Compose die Containernamen aus dem Namen des Verzeichnisses, in dem die Compose-Datei liegt, aus dem Namen des Services, der in der Datei `'docker-compose.yaml'`eingetragen wurde und einer laufenden

Nummer. Die drei Strings werden jeweils durch einen Unterstrich miteinander verbunden.

Mit dem `attach`-Kommando verbinden wir uns mit dem Container und landen auf dessen Kommando-Shell.

```
1  > docker container attach doubleservice_ubuntu_ext_1
```

Hier können Sie noch einmal die Kommandos `ping` und das `curl` testen.

Die Ergebnisse der obigen Eingaben sollte so wie im folgenden Screenshot einer PowerShell aussehen (Abb. 12.8).

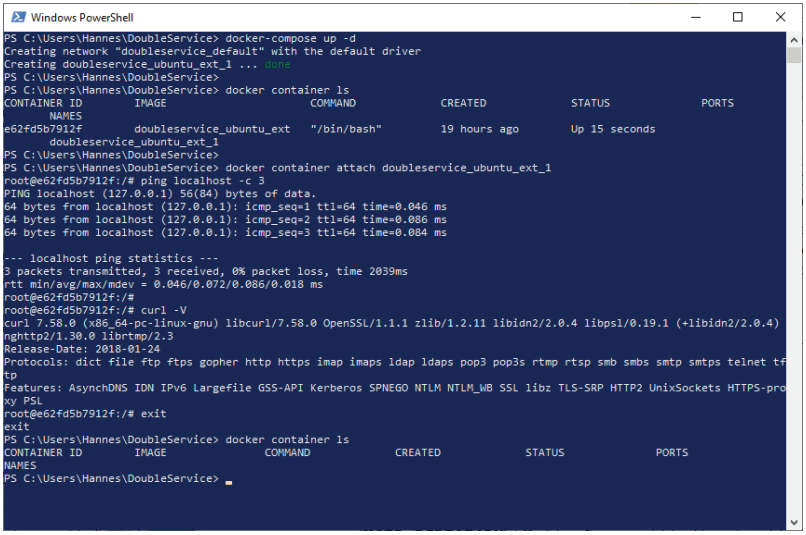



**Abb. 12.8** Test des erweiterten Ubuntu Containers mit `Docker Compose`

### 12.8.3 Einbinden eines NGINX Containers über Docker Compose

In einem letzten Schritt erweitern wir unseren Service um einen NGINX Container und aktivieren ein Netzwerk, das die beiden Container verbindet.

Da auch hier unser NGINX Container unsere eigene Webseite anzeigen soll, kopieren wir das Verzeichnis `html` mit der Datei `index.html` aus dem Beispiel „Hello-Compose“ in das Verzeichnis `DoubleService`, welches den Kontext für unseren Service darstellt:

Verzeichnis

```
<USER_DIR>\Hello-Compose\html\
```

nach

```
<USER_DIR>\DoubleService\html\
```

kopieren.



Jetzt erweitern wir im Verzeichnis `<USER_DIR>\DoubleService` die Docker-Compose-Datei gemäß dem folgenden Beispiel:

```
Datei ,docker-compose.yaml'
version: "3.7"
services:

    ubuntu_ext:
        build: ../UbuntuExt
        stdin_open: true
        tty: true
        networks:
            - test_net

    my_web:
        image: nginx:1.17.7
        ports:
            - "8080:80"
        volumes:
            - ./html:/usr/share/nginx/html
        networks:
```

```
19              - test_net
20
21  networks:
22      test_net:
```

Es wurde hier am Ende die Sektion `networks` angefügt und darunter der Netzwerkname `test_net` definiert.

In der sektion `ubuntu_ext` von `services` wurde ebenfalls die Sektion `networks` mit dem Netzwerknamen `test_net` eingefügt. Damit hat jetzt der Container `doubleservice_ubuntu_ext_1` Zugriff auf dieses Netzwerk.

Weiterhin haben wir im Beispiel die Sektion `services` um den Eintrag für einen NGINX Service mit dem Namen `my_web` erweitert. Das entspricht dem Inhalt der Compose-Datei aus Kapitel 12.6 und wird ebenfalls durch die Sektion `networks` ergänzt, damit auch dieser Container über das Netzwerk `test_net` kommunizieren kann.

Testen wir jetzt unseren neuen Service mit zwei Containern:

Wir starten eine Kommando-Shell und wechseln in das Verzeichnis `<USER_DIR>\DoubleService`. Das ganze wird, wie gehabt, mit Docker Compose gestartet:

```
1  > docker-compose up -d
```

Sehen wir jetzt einmal nach, ob der neue Netzwerkname auch in der Liste der Netzwerke angezeigt wird (Abb. 12.9):

```
1  > docker network ls
```

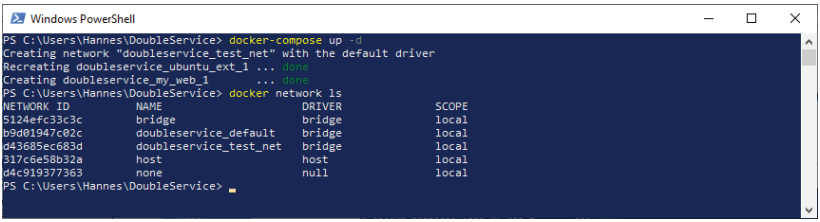



**Abb. 12.9** Die Netzwerke aus der Übung „DoubleService“

Auch hier wird der Netzwerkname aus dem Namen des Kontexts und dem Netzwerk-Namen, der in der Compose-Datei eingetragen wurde, zusammengesetzt und lautet `doubleservice_test_net`.

Das Netzwerk mit dem Namen `doubleservice_default` ist das Standardnetzwerk, welches automatisch erstellt wird, ohne besondere Einträge in der Compose-Datei.

Mehr Informationen zu unserem neuen Netzwerk erhalten wir vom Inspect-Kommando (Abb. 12.10):

```
1  > docker network inspect doubleservice_test_net
```

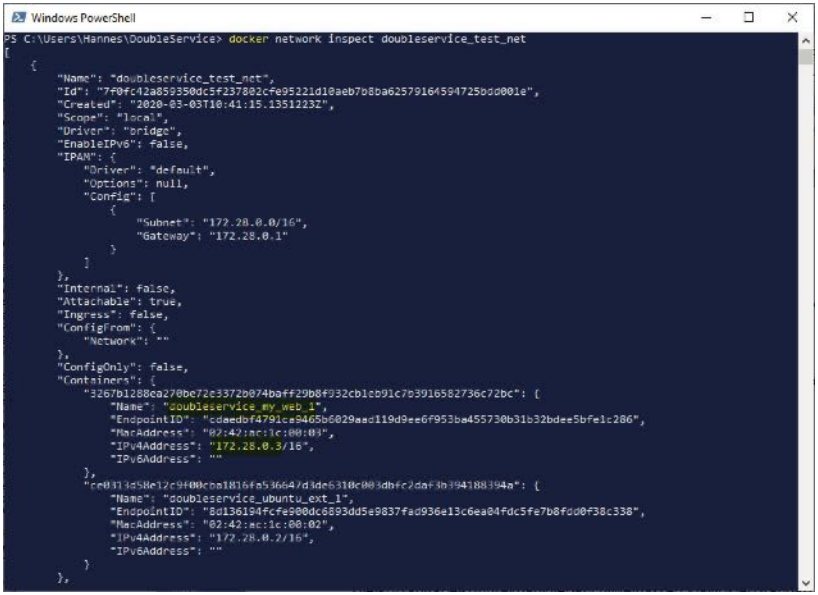



**Abb. 12.10** Informationen über das `doubleservice_test_net` Netzwerke aus der Übung „DoubleService“

Wir merken uns hier von diesen Informationen die IP des Containers `doubleservice_my_web_1` (hier 172.28.0.3).

Verbinden wir uns jetzt mit dem Ubuntu Container.

```
> docker container attach doubleservice_ubuntu_ext_1
```

Am Kommando Prompt des Ubuntu Containers geben wir zuerst ein `ping`-Kommando mit der IP unseres NGINX Containers ein:

```
# ping 172.28.0.3 -c 3
```

Ersetzen Sie bei diesem Kommando die IP mit dem Wert, der in Ihrem Fall von Docker vergeben wurde.

Anschließend ersetzen wir beim `ping`-Kommando die IP durch den Containernamen `doubleservice_my_web_1`:

```
# ping doubleservice_my_web_1 -c 3
```

Als Nächstes wollen wir testen, ob wir mit `curl` die Webseite aus dem NGINX Container abfragen können. Zuerst geben wir dabei die URL mit der IP-Adresse an:

```
# curl http://172.28.0.3:80
```

Sie müssen die IP wieder durch die ersetzen, welche bei Ihnen vergeben wurde.

Hier der Screenshot mit den Ausgaben der im obigen Beispiel angegebenen Kommandos (Abb. 12.11).

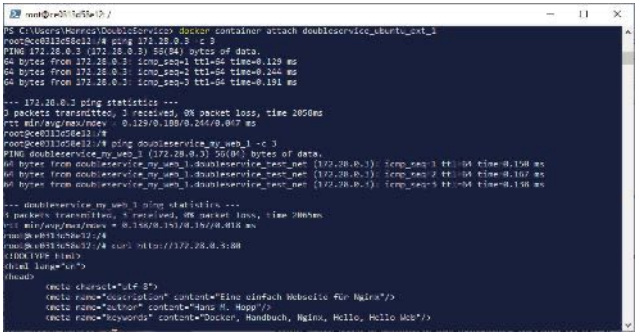



**Abb. 12.11** Test der Netzwerk-Kommunikation zwischen Containern aus „DoubleService“

Zuletzt führen wir den curl-Test durch, indem wir den Containernamen als URL angeben (Abb. 12.12):

```
1  # curl http://doubleservice_my_web_1:80
```

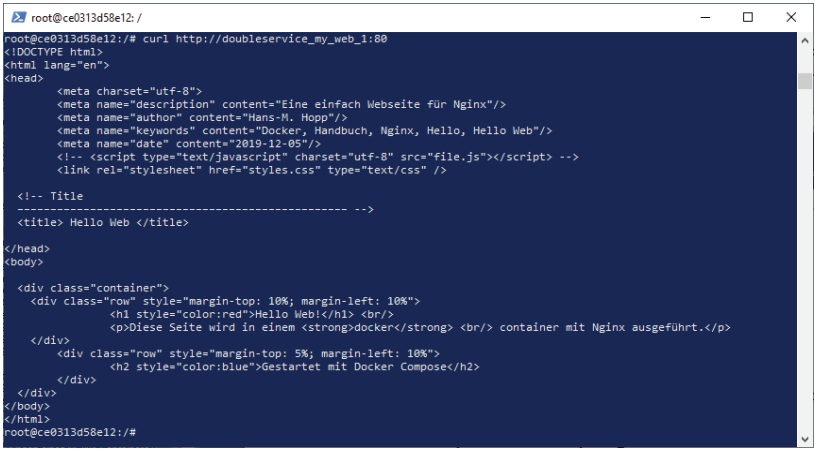



**Abb. 12.12** Test des `curl`-Kommandos in „DoubleService“ mit einem Containernamen als URL



Die Portangabe am Ende der URL könnte man in diesem Fall auch weglassen, da 80 der Standartport ist.

Zum Beenden des Attach-Modes geben Sie das Kommando `# exit` ein.

**Achtung!**

Damit wird auch der gesamte Container `doubleservice_ubuntu_ext_1` beendet.

Beenden Sie die restlichen, noch laufenden Container aus dem Service „DoubleService“:

```
1  > docker-compose down
```

Zu Sicherheit überprüfen wir noch einmal, ob tatsächlich alle Container beendet wurden, indem wir alle noch laufenden Container auflisten (Abb. 12.13):

```
> docker container ls
```

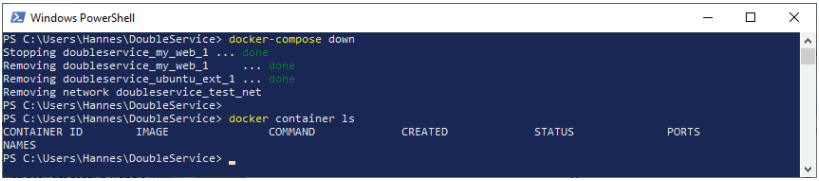



**Abb. 12.13** Beenden von „DoubleService“

## 12.9 Umgebungsvariablen nutzen

### 12.9.1 Umgebungsvariable in einer Datei

Docker Compose erlaubt es, Umgebungsvariablen über eine Datei zu deklarieren. Docker Compose sucht diese Datei im aktuellen Verzeichnis, also in dem Verzeichnis, in dem Docker Compose gestartet wurde.

Docker Compose sucht standardmäßig nach einer Datei mit dem folgenden Namen:

```
,'.env''
```

Für den Inhalt dieser Datei gelten die folgenden Regeln:

- Kommentare werden durch das Hash-Zeichen (#) am Zeilenanfang gekennzeichnet. Diese Zeilen werden ignoriert.
- Leerzeilen werden ignoriert.
- Pro Zeile ist eine Variablendeklaration möglich.
- Die Deklaration hat die Form <VARIBLE>=<WERT>
  Beispiel: DEBUG=1

- Anführungszeichen sind Teil des zugewiesenen Wertes und werden nicht gesondert behandelt.

Die Werte der Variablen aus der Datei können zur Laufzeit überschrieben werden.

Umgebungsvariablen, die in der Datei ,'.env'' deklariert wurden, sind nicht automatisch in den Containern sichtbar.

### 12.9.2 Umgebungsvariablen in Compose

Mit Docker Compose ist der Zugriff auf Umgebungsvariablen möglich, die in der aktuellen Shell angelegt wurden. Will man den Inhalt einer Umgebungsvariablen nutzen, so wird dies in der Compose-Datei wie folgt angegeben:

```
1 ${<VARIABLEN_NAME>}
```

Man könnte zum Beispiel die Version eines Images einer Umgebungsvariablen der Shell zuweisen und in der Compose-Datei übernehmen.



Angenommen, in einer Shell wurde eine Umgebungsvariable `NGINX_VERSION` mit dem Wert `1.17.7` angelegt:

```
1 NGINX_VERSION=1.17.7
```

Der Wert dieser Variablen kann danach in der Compose-Datei wie folgt übernommen werden:

```
1 services:
2     my_web:
3         image: "nginx:${NGINX_VERSION}"
```

Ist die Umgebungsvariable nicht gesetzt, wird eine leere Zeichenkette eingesetzt. Allerding kann man diese Variable in der Datei `'.env'` anlegen. Der dort zugewiesene Wert wird dann als Standardwert eingesetzt. Wie schon oben erwähnt, werden Variablen aus der `'.env'` Datei aber überschrieben, wenn diese in der aktuellen Shell definiert wurden.

Falls sie ganz sicher gehen wollen, dass eine Umgebungsvariable nicht in einen leeren String aufgelöst wird, dann kann zusätzlich noch ein sogenannter Inline Default-Wert angegeben werden. Der wird durch einen Doppelpunkt, Bindestrich oder beides getrennt, an den Namen der Umgebungsvariablen angehängt:

```
image: "nginx:${NGINX_VERSION:-latest}"
```

Die oben gezeigte Form löst die Variable in den Wert *latest* auf, wenn die Variable leer oder nicht gesetzt ist.

Hier noch eine andere Form:

```
image: "nginx:${NGINX_VERSION-latest}"
```

Mit dieser Form wird die Variable nur dann in den Wert *latest* aufgelöst, wenn die Variable nicht gesetzt ist.

```
image: "nginx:${NGINX_VERSION:latest}"
```

Mit dieser Form wird die Variable nur dann in den Wert *latest* aufgelöst, wenn die Variable leer ist.

### 12.9.3 Umgebungsvariablen in Containern

Will man Docker-Compose-Umgebungsvariablen verwalten, die innerhalb eines Containers genutzt werden sollen, so wird dies mit dem Key `environment` unterhalb der zugehörigen `service`-Sektion angegeben.

```
services:
    ubuntu_ext:
        …
        environment:
            - DATABASE=my_db
            - BRANCH=develop
```

Die obige Form nutzt die Array-Notation von YAML. Es kann aber als Alternative auch die *Dictionary-Notation* (<KEY>:<VALUE>) verwendet werden.

```
services:
    ubuntu_ext:
        …
        environment:
            DATABASE: my_db
            BRANCH: develop
```

Sollen die Werte von Umgebungsvariablen aus der Shell oder aus der Datei `'.env'` an einen Container einfach weitergereicht werden, ohne diese zu verändern oder neu zu setzen, dann gibt man nur den Namen der Umgebungsvariablen an, ohne eine Wertzuweisung.

```
services:
    ubuntu_ext:
        …
        environment:
            - DATABASE
            - BRANCH
```

Auch hier kann die Angabe in der *Dictionary*-Form genutzt werden:

```
services:
    ubuntu_ext:
        …
        environment:
            DATABASE:
            BRANCH:
```

### 12.9.4 Übungsaufgabe: Einsatz von Umgebungsvariablen

Erstellen Sie eine Docker-Compose-Datei, die einen Container aus einem Ubuntu Image erstellt. Die Version des Images soll aus einer Umgebungsvariablen mit dem Namen `UBUNTU_VERSION` übernommen werden.

Gibt es die Variable `UBUNTU_VERSION` nicht oder ist sie leer, dann soll `latest` als Image genutzt werden.

Diese Variable soll in der Datei `'.env'` deklariert werden.

Im Ubuntu Container soll es außerdem eine Umgebungsvariable mit dem Namen `USER_NAME` geben. Diese soll den Wert einer Umgebungsvariablen aus der aufrufenden Shell übernehmen. Gibt es die Variable `USER_NAME` nicht oder ist sie leer, dann soll sie den Wert `unknown` erhalten. Auch diese Variable soll über die Datei `'.env'` deklariert werden.

Der Container soll im interaktiven Modus mit einem Pseudo TTY gestartet werden, damit wir die Ergebnisse überprüfen können.

***Lösung:***

Für das Lösungsbeispiel wurde unter dem Benutzerverzeichnis ein Verzeichnis mit dem Namen `'EnvironmentTest'` erstellt. Dort befinden sich die beiden Dateien `'docker-compose.yaml'` und `'.env'`.

Beispiel für die Docker-Compose-Datei

```
Datei ,docker-compose.yaml'
version: "3.7"
services:

    ubuntu:
        image: "ubuntu:${UBUNTU_VERSION:-latest}"
        stdin_open: true
        tty: true
        environment:
            - USER_NAME=${USER_NAME:-unknown}
```

Beispiel für die Environment-Datei:

```
Datei ,".env"
UBUNTU_VERSION=14.04
USER_NAME=HUGO HOPPER
```



Start des Containers:

```
> docker-compose up -d
```

An den Container anbinden:

```
> docker container attach environmenttest_ubuntu_1
```

Version in der Ubuntu Shell anzeigen lassen:

```
# lsb_release –a
```

oder

```
# cat /etc/lsb-release
```

Den Inhalt der Umgebungsvariablen `USER_NAME` anzeigen lassen:

```
1  # echo $USER_NAME
```

Hier ein Screenshot der PowerShell mit diesem Übungsbeispiel (Abb. 12.14):

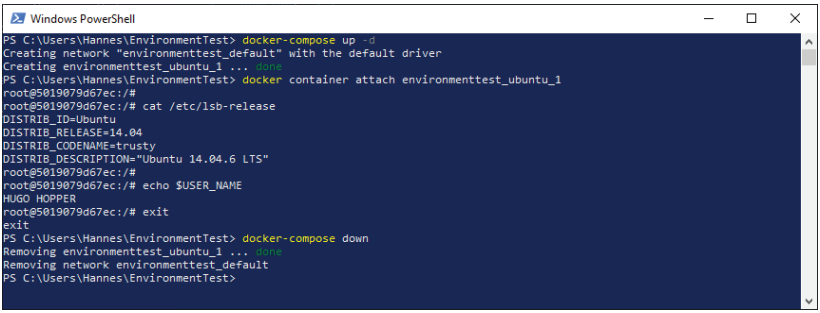



**Abb. 12.14** Übungsaufgabe ‚Einsatz von Umgebungsvariablen'

## 12.10 Services skalieren

Unter dem Begriff „Skalierbarkeit" versteht man im Allgemeinen die Fähigkeit von Systemen, ihre Größe bzw. ihre Ressourcen an die aktuellen Anforderungen anzupassen.

Bei Microservices bedeutet dies, dass je nach Bedarf mehrere Instanzen eines Service gleichzeitig gestartet werden können, um damit beispielsweise auf die zunehmende Last eines Systems reagieren zu können.

Beim Start eines Containers mit dem `up`-Kommando kann ein Skalierungsfaktor für die zu startenden Services per Parameter angegeben werden.

Beipiel:

```
1  > docker-compose up --scale redis-master=3 -d
```

Docker Compose bietet für Versionen 2.x die Möglichkeit, einen Skalierungsfaktor für Container in die Compose-Datei einzutragen. Dies wird unter dem zugehörigen Service-Eintrag mit dem Key `scale` angegeben.

Beim folgenden Beispiel wird unser Beispiel DoubleService so erweitert, dass vom Service `my_app` drei Instanzen erzeugt werden:

```
version: '2.3'
    services:
    my_app:
            build: .
            image: „example-app
            …
            scale: 3
            …
```

Wurde in der Compose-Datei ein `scale`-Wert für einen Container definiert, dann wird dieser mit dem `–scale` Flag des `docker-compose up`-Kommandos überschrieben.

Achtung: Bei Versionen 3.x von Docker Compose produziert ein `scale`-Eintrag in der Datei `docker-compose.yaml` eine Fehlermeldung nach dem Start der Services mit dem Kommando `docker-compose up`: "Unsupported config option for services.my_app: 'scale'"



Eine alte Variante, Services zu skalieren, stellt das Kommando `scale` von `docker-compose` dar. Die Anwendung dieses Kommandos wird aber nicht mehr empfohlen. Der Vollständigkeit halber stellen wir es hier aber noch kurz vor.

Hier die Syntax:

```
docker-compose scale <SERVICE_NAME>=<ANZAHL>
```

Hier noch ein Beispiel:

```
1 > docker-compose scale redis-master=3
```

Mehr zum Thema Skalierung von Services erfahren Sie in den Kapiteln über Docker Swarm bzw. Kybernetes.

## 12.11 Log-Dateien

Container, die durch das Docker-Compose-Kommando `up` gestartet wurden, produzieren natürlich auch Log-Informationen.

Man könnte nun die Logs für jeden Container einzeln durch das bereits bekannte Kommando

```
1 docker logs <CONTAINER_NAME>
```

abrufen.

Da aber in der Praxis sehr viele Services als Container gestartet werden können, wäre die Abfrage der Log-Dateien für jeden einzelnen Container schnell unübersichtlich und auch mühselig.

Aus diesem Grund stellt uns Docker Compose ein eigenes `logs`-Kommando zur Verfügung, mit dem die Log-Daten aller Services ausgegeben werden, die in einer Compose-Datei definiert wurden und über das Kommando `docker-compose up` gestartet wurden.

Dazu startet man eine Kommando-Shell, wechselt in das Verzeichnis, das die entsprechende Docker-Compose-Datei enthält und gibt das folgende Kommando ein:

```
1 > docker-compose logs
```

Wie schon beim Kommando `docker logs`, gibt es auch hier den Parameter `-f`, um kontinuierliche Log-Ausgaben zu erhalten.

Um das auch in der Praxis zu sehen, führen wir die folgenden Aktionen durch:

- Starten Sie eine Shell.
- Wechseln Sie in das Verzeichnis mit dem Beispiel „DoubleService“:
  `> cd .\DoubleService\`
- Starten Sie die Services:
  `> docker-compose up -d`
- Damit Log-Einträge generiert werden, starten Sie einen Browser und verbinden sich mit der Webseite aus dem NGINX Container. Geben Sie im Adressfeld des Containers die folgende URL ein: http://localhost:8080
- Um die Log-Einträge auszugeben, müssen Sie jetzt nur noch `docker-compose` mit dem `logs` Kommando aufrufen:
  `> docker-compose logs`
- Zuletzt beenden wir die Services:
  `> docker compose down`

Der nächste Screenshot zeigt die Ausführung der oben angegebenen Aktionen in einer PowerShell (Abb. 12.15).

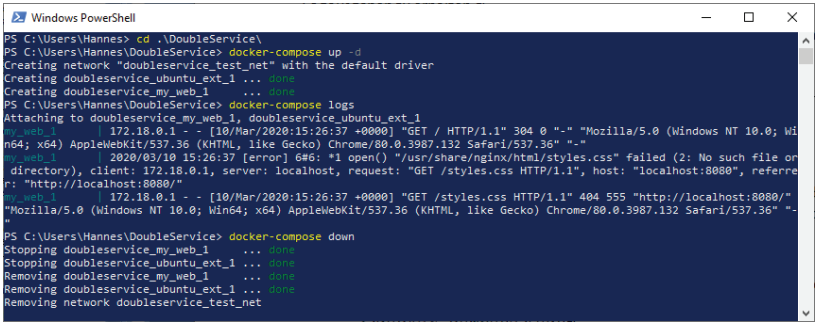



**Abb. 12.15** Beispiel für Log-Ausgaben mir Docker Compose

Kapitel 13

# Wordpress-Blog mit Docker Compose

In Kapitel 11 dieses Buches haben wir eine Docker-Umgebung für einen WordPress-Blog aufgesetzt, indem wir die benötigten Docker Container manuell gestartet haben.

Wie Sie dabei sicher schon erkannt haben, ist der manuelle Start von mehreren voneinander abhängigen Docker Containern mit etlichen Nachteilen verbunden.

Es muss hier beim Start eine umfangreiche und komplexe Liste von Parametern eingegeben werden, wobei sich da sehr leicht Fehler einschleichen können.

Wenn so ein System dann auch noch an Kollegen oder Kunden weitergegeben werden soll, die den Start der Services ausführen sollen, dann müssen diese neben den Angaben zu den Kommandozeilenparameter auch noch Informationen über die Reihenfolge der Ausführung von Kommandos, Vorbedingungen und sonstige Informationen erhalten.



Sie haben es längst erkannt – das ist ein klassischer Anwendungsfall für den Einsatz von Docker Compose.

Bevor wir beginnen, erstellen wir ein neues Verzeichnis für unser WordPress-Projekt:

Legen Sie ein Verzeichnis mit dem Namen `'WordPress'` unter Ihrem Benutzerverzeichnis an und wechseln Sie in dieses Verzeichnis.

Erstellen Sie dort die folgenden Unterverzeichnisse: `plugins`, `themes` und `uploads`.

Erzeugen Sie im Verzeichnis `'WordPress'` eine Datei `'docker-compose.yaml'` und tragen Sie dort die folgenden Anweisungen ein:

```
Datei 'docker-compose.yaml'

version: "3.7"

services:
    database:
        image: mysql:5.7
        volumes:
            - database_data:/var/lib/mysql
        restart: always
        environment:
            MYSQL_ROOT_PASSWORD: mypassword
            MYSQL_DATABASE: wordpress
            MYSQL_USER: wordpress
            MYSQL_PASSWORD: wordpress

    wordpress:
        image: wordpress:latest
        depends_on:
            - database
        ports:
            - 8080:80
        restart: always
        environment:
            WORDPRESS_DB_HOST: database:3306
            WORDPRESS_DB_USER: wordpress
            WORDPRESS_DB_PASSWORD: wordpress
        volumes:
            - .wordpress/plugins:/var/www/html/wp-content/plugins
            - .wordpress/themes:/var/www/html/wp-content/themes
            - .wordpress/uploads:/var/www/html/wp-content/uploads
volumes:
    database_data:
```

In der ersten Zeile geben wir die Version von Docker Compose an.

Anschließend definieren wir die Container Services.

Der Datenbank-Service erhält den Namen "database". Dieser muss auch hier auf dem Image MySQL 5.7 aus dem Docker Hub basieren.

In das Container Volume mit dem Namen `'database_data'` mappen wir das Standard-Verzeichnis von MySQL - `/var/lib/mysql`.

Schließlich definieren wir die benötigten Environment-Variablen für den Datenbank-Container.

Natürlich wählen Sie für Ihre Container Ihre eigenen Passwörter und diese sind dann selbstverständlich so sicher, dass kein Hacker dieser Welt sie knacken kann.

In der nächsten Sektion legen wir die Eigenschaften für den WordPress Container fest.

Mit dem Schlüsselwort `"depends_on"` geben wir an, dass der WordPress Container vom Datenbank-Container abhängig ist. Damit wird er erst gestartet, wenn der Start des „MySQL" Services abgeschlossen ist.

Als Image geben wir die aktuellste Version von WordPress im Docker Hub an.

Außerdem mappen wir Port 8080 aus dem Host auf Port 80 im Container.



InderEnvironment-Variable"`WORDPRESS_DB_HOST: database:3306`" teilen wir WordPress mit, dass der Container `database` die SQL-Datenbank für WordPress enthält. Der Port 3306 stellt den Standard-Port für Verbindungen zum MySQL Container dar.

Ansonsten weisen wir auch hier wieder unseren Usernamen und das Passwort für die Datenbank den entsprechenden Environment-Variablen zu.

Unsere Verzeichnisse `plugins, themes` und `uploads` mappen wir zuletzt auf die zugehörigen Container Volumes.

Mit der Anweisung `restart: always` legen wir fest, dass ein Container nach einem Stopp wieder automatisch gestartet wird. Nach einem

manuellen Stop oder nach einem Fehler beim Container-Start wird aber kein Neustart ausgeführt. Damit wird verhindert, dass ein Container in einer Endlosschleife immer wieder gestartet wird.

Wir starten die Services mit Docker Compose wie gewohnt:

```
1  > docker-compose up -d
```

Es dauert wieder eine ganze Weile, bis die Container starten.

Sehen wir jetzt noch einmal nach, ob die beiden Container wirklich gestartet wurden (Abb. 13.1):

```
1  > docker container ls
```

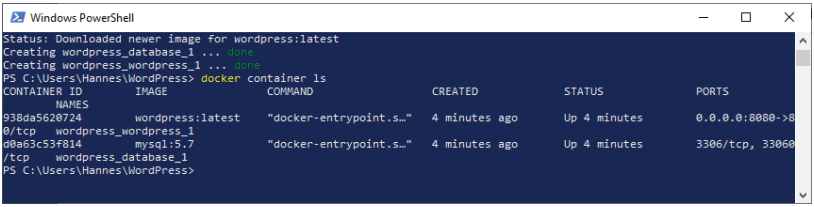



**Abb. 13.1** Start von WordPress mit Docker Compose

Abschließend testen wie noch, ob wir wieder auf unsere WordPress-Applikation über den Webbrowser zugreifen können.

Hier noch einmal die URL für die Adresszeile des Browsers:

http://localhost:8080/wp-admin/install.php

Und wieder sollten wir auf der ersten Seite der WordPress-Installationsroutine landen und können WordPress noch einmal für Ihre Containeranwendung installieren, um anschließend damit eine Webseite zu gestalten.

Viel Spaß beim Erstellen Ihrer Webseite!

Kapitel 14

# Datenbank im Container

Datenbanken sind heutzutage ein wichtiger Bestandteil von Applikationen, egal ob es sich dabei um herkömmliche, lokale Programme oder um Web-Anwendungen handelt. Wenn Sie zukünftig in Ihren eigenen Docker-Anwendungen Daten speichern müssen, dann werden Sie um den Einsatz von Datenbanken wahrscheinlich nicht herumkommen.

Aus diesem Grund werden wir in diesem Kapitel den Einsatz von Datenbanken in Verbindung mit Docker Containern etwas genauer betrachten.

## 14.1 Beispiel MariaDB mit phpmyadmin

Den Einsatz eines Docker Containers zur Verwaltung einer MySQL-Datenbank haben wir ja bereits in den Kapiteln zu Erstellung einer WordPress-Applikation kennengelernt.

Hier stellen wir Ihnen den Einsatz eines anderen Datenbank-Containers vor, der auf dem Image für MariaDB aus dem Docker Hub basiert.

Das Image von MariaDB wurde bereits im Kapitel 4.4 kurz vorgestellt. Wie dort erwähnt, handelt es sich bei MariaDB um eine Abspaltung während der Entwicklung von MySQL.

Ein MariaDB Container kann als eine Datenbank-Alternative zu MySQL eingesetzt werden. Wir verwenden in diesem Beispiel MariaDB, um den Umgang mit diesem Image vorzustellen. Sie werden dabei feststellen, dass es in der Praxis eigentlich nicht viele Unterschiede zwischen den beiden Datenbanksystemen MySQL und MariaDB gibt.

Unsere WordPress-Blog-Anwendung hätte übrigens ebenso gut mit MariaDB als Datenbank funktioniert. Wir haben bei der Entwicklung

unserer WordPress-Applikation bereits einen Datenbank-Container mit WordPress kombiniert. Den Inhalt der Datenbank hat dort WordPress selbst verwaltet.

Jetzt wollen wir die Daten in einer Datenbank selbst erstellen und verwalten. Dafür werden wir einen Container mit dem Image `phpmyadmin` aus dem Docker Hub nutzen.

Bei phpMyAdmin handelt es sich um eine freie Web-Anwendung, die zur Verwaltung von MySQL und MariaDB-Datenbanken entwickelt wurde. Diese Anwendung erlaubt es uns, eine SQL-Datenbank zu bearbeiten, ohne dass wir dafür SQL-Anweisungen eingeben müssen (falls Sie aber die SQL-Kommandos selber eingeben wollen, dann ist das mit diesem Tool ebenfalls möglich).

Da dieses Buch das Thema Docker behandelt und SQL nicht direkt unser Thema ist, bearbeiten wir hier unsere Datenbank mithilfe dieser Applikation.

Für das Datenbank-Beispiel legen wir wieder ein neues Verzeichnis an. Sie können es unter Ihrem Benutzerverzeichnis mit dem Namen `'MariaDB'` anlegen. Falls Sie einen anderen Pfad wählen wollen, dann müssen Sie die Beispiele entsprechend anpassen.



```
<USER_DIR>\MariaDB
```

In das neue Verzeichnis kommt die Docker-Compose-Datei für unser Beispiel. Der Inhalt von dieser Datei soll folgendermaßen aussehen:

```
Datei 'docker-compose.yaml'

version: "3.7"
services:

    database:
        image: mariadb:latest

        restart: always

        volumes:
```

```
12              - mariadb_data:/var/lib/mysql
13
14          environment:
15              MYSQL_ROOT_PASSWORD: topsecret
16              MYSQL_DATABASE: telefon
17              MYSQL_USER: user
18              MYSQL_PASSWORD: topsecret
19
20      phpmyadmin:
21
22          image: phpmyadmin/phpmyadmin
23
24          ports:
25              - "8080:80"
26
27          restart: always
28
29          environment:
30              PMA_HOST: database
31
32          depends_on:
33              - database
34
35  volumes:
36      mariadb_data:
```

Die erste Zeile dieser Docker-Compose-Datei gibt an, dass die Syntax der Version 3.7 von Docker Compose verwendet wird.

In der `Container Services`-Sektion definieren wir den Service "`database`" für die MariaDB-Datenbank und den Service "`phpmyadmin`" für die Datenbank-Verwaltungsapplikation.

Für beide Container geben wir als Restart Policy „`always`" an. Damit werden beide Container nach einem Stopp wieder automatisch gestartet (nicht nach einem manuellen Stopp).

Der Datenbank-Service basiert in unserem Beispiel auf der aktuellsten Version des MariaDB Images.

In das Container Volume mit dem Namen ‚`mariadb_data`' mappen wir das Standard-Verzeichnis von MariaDB – auch hier ist dies. `'/var/lib/mysql'.`

Es müssen wieder die Werte der Environment-Variablen für den Datenbank-Container gesetzt werden.

Für den Service `phpMyAdmin` legen wir Folgendes fest:

Als Image wird die aktuellste Version von „**phpmyadmin**“ aus dem Docker Hub verwendet.

Port 8080 aus dem Host wird auf Port 80 im Container gemappt.

Über die Environment-Variable `PMA_HOST: database` teilen wir phpMyAdmin mit, dass der Container `database` die SQL-Datenbank für phpMyAdmin enthält. Da der Port 3306 der Standard-Port für Verbindungen zu MySQL Containern ist, wird er von phpMyAdmin automatisch verwendet und wir können diese Angabe hier auch weglassen.

Mit dem Eintrag `depends_on` geben wir an, dass der phpMyAdmin Container vom Datenbank-Container abhängig ist und erst gestartet werden kann, wenn der Start vom Datenbank-Container erfolgt ist.

Wir starten die Services über Docker Compose:

```
1  > docker-compose up -d
```

Sehen wir jetzt noch einmal nach ob die beiden Container wirklich gestartet wurden:

```
1  > docker container ls
```

Wenn beide Container laufen, dann können wir die neue Anwendung testen.

Starten Sie einen Internet-Browser. Als URL geben Sie wieder einmal ‚`http://localhost`‘ mit der Portnummer 8080 an:

http://localhost:8080/

Und voila: Die Anmeldeseite von phpMyAdmin erscheint (Abb. 14.1):


**Abb. 14.1** Die Anmeldeseite von „phpMyAdmin“

Wählen Sie hier die von Ihnen gewünschte Sprache aus.

Geben Sie dann den Benutzernamen „root“ und das Passwort „topsecret“ ein, um sich als Admin anzumelden. Wollen Sie sich als einfacher Benutzer anmelden, dann geben Sie „user“ (oder den von Ihnen in `'docker-compose.yaml'` verwendeten Namen) ein. Klicken Sie den [OK]-Button, um sich auf der phpMyAdmin-Webseite anzumelden. Falls Sie in der Datei `'dockercompose.yaml'` der Environment-Variable `MYSQL_ROOT_PASSWORD` beziehungsweise `MYSQL_PASSWORD` ein anderes Passwort als `'topsecret'` zugewiesen haben, dann müssen Sie natürlich jenes verwenden.

Nach dem erfolgreichen Login wird die Startseite von phpMyAdmin geöffnet (Abb. 14.2).

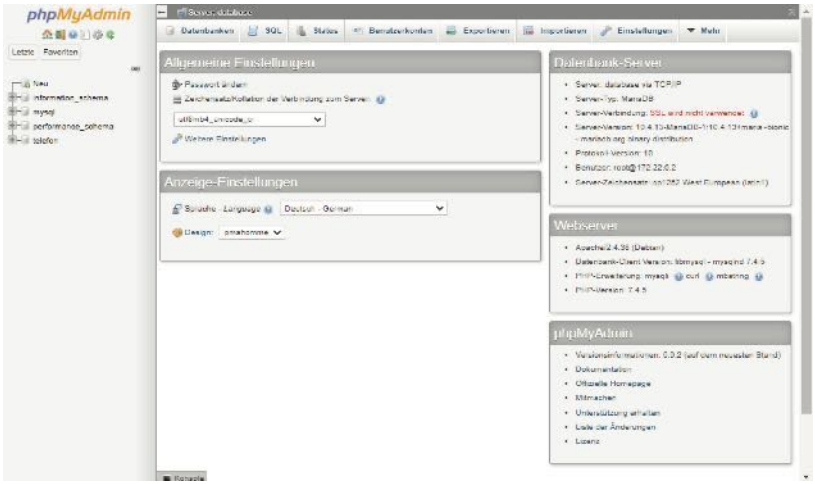



**Abb. 14.2** Die Startseite von „phpMyAdmin“

Hier werden Ihnen bereits zahlreiche Informationen zur Verfügung gestellt. Der obige Screenshot zeigt im Feld DATENBANK-SERVER, dass der Datenbank-Server vom Typ MariaDB ist und dass als Webserver Apache in der Version 2.4.38 läuft. Die Seite erlaubt es auch, das Passwort oder die Sprache zu ändern. Falls Ihnen das aktuelle Design der Seite nicht gefällt, dann können Sie bei den Anzeige-Einstellungen über das Listenfeld „DESIGN“ auch ein anderes auswählen.



An dieser Stelle soll jetzt keine vollständige Einführung in phpMyAdmin gegeben werden. Es wird Ihnen lediglich gezeigt, wie mit diesem Tool eine Datenbank mit ein paar einfachen Daten gefüllt werden kann. Falls Sie ausführlichere Informationen wünschen, dann klicken Sie im Feld PHPMYADMIN rechts unten auf den Eintrag *Dokumentation*. Damit öffnen Sie das Handbuch der Applikation.

Auf der Startseite wird in der linken Spalte eine Baumstruktur angezeigt. Hier befindet sich auch ein Element für die Datenbank mit dem Namen `'telefon'`. Diesen Namen haben wir in der Datei `'docker-compose.yaml'` mit dem Eintrag `MYSQL_DATABASE: telefon` festgelegt.

Durch Mausklick auf dieses Element öffnen Sie das Strukturfenster für diese Datenbank. Hier werden Ihnen unter anderem Steuerelemente angeboten, über die Sie für diese Datenbank eine neue Tabelle anlegen können (Abb. 14.3).

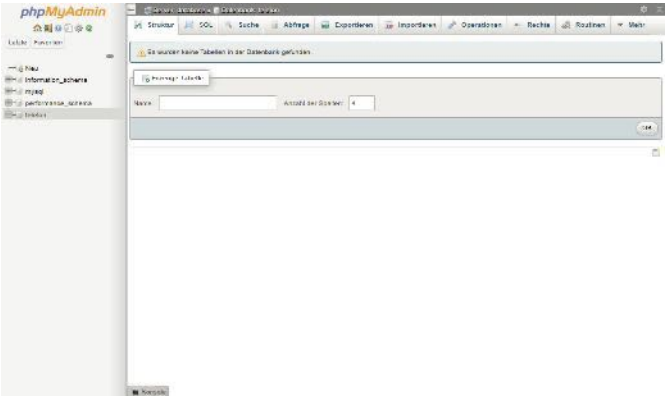



**Abb. 14.3** Die Strukturseite von „phpMyAdmin“

Um eine neue Tabelle zu erstellen, geben Sie auf dieser Seite im Feld *Name:* den Tabellennamen `'Telefonliste'` ein, stellen bei Anzahl der Spalten 2 ein und danach klicken Sie rechts auf die Schaltfläche [OK].

Jetzt wird ein Fenster angezeigt, mit dem man die Felder der neuen Tabelle spezifizieren kann (Abb. 14.4).

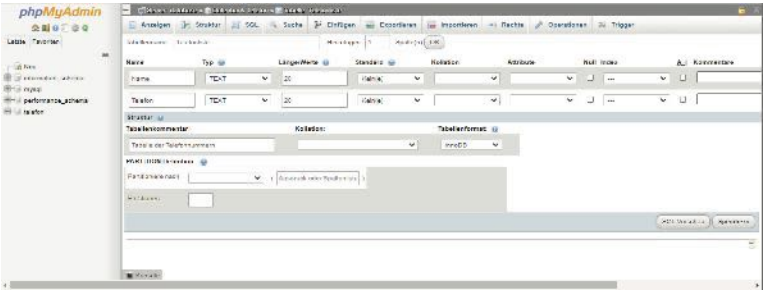



**Abb. 14.4** Bearbeitung von Tabellenfeldern auf der Strukturseite von „phpMyAdmin“

Übernehmen Sie die Einträge für die beiden Spalten aus dem Beispiel im Screenshot. Wenn Sie wollen, können Sie für diese Angaben auch eine Vorschau des SQL-Befehls anzeigen lassen. Das geht über die Schaltfläche [SQL Vorschau].

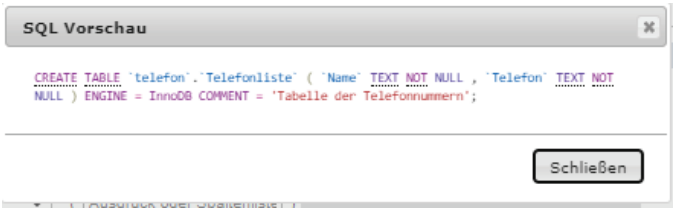



**Abb. 14.5** SQL-Vorschau von „phpMyAdmin“

Beenden Sie die Bearbeitung der Tabelle durch einen Klick auf den Button [Speichern] rechts unten.

Nun fehlen nur noch ein paar Daten in unserer Datenbank. Um die zu erstellen, klicken wir in der oberen Leiste auf das Register [Einfügen] (Abb. 14.6).

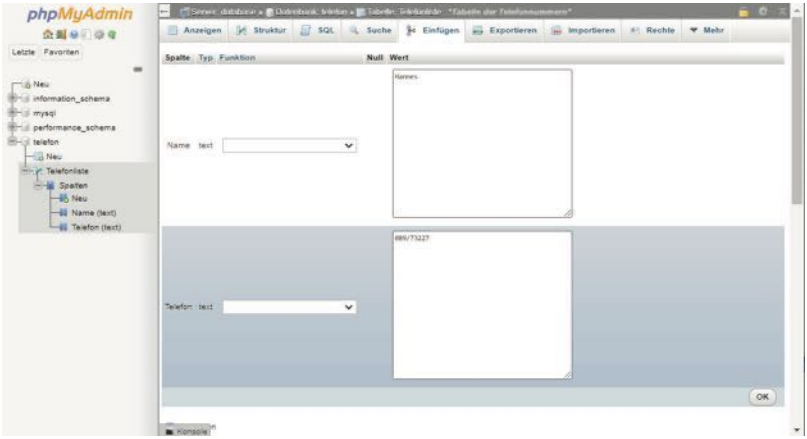



**Abb. 14.6** Die Einfügenseite von „phpMyAdmin“

Geben Sie hier die gewünschten Daten für die Felder ein und übergeben Sie jeden neuen Datensatz jeweils mit OK an die Tabelle.

Dabei wechselt die Ansicht jedesmal zum Register [SQL], wo der gerade ausgeführte SQL-Befehl zur Information angezeigt wird (Abb. 14.7).

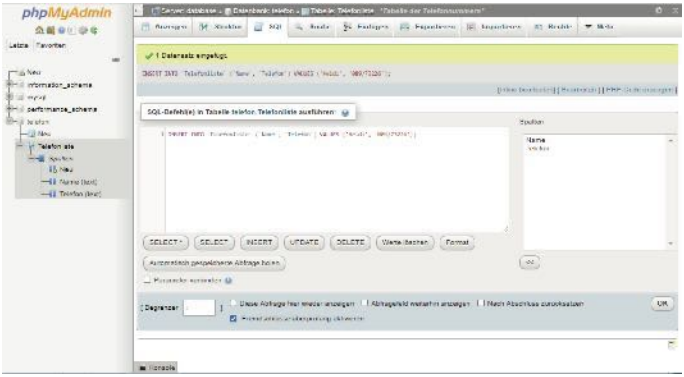



**Abb. 14.7** Die SQL-Seite von „phpMyAdmin“

Wechseln Sie wieder auf das Register [Einfügen], um weitere Einträge zu erzeugen.

Sie wollen sicher auch einmal eine Übersicht über die aktuellen Einträge einsehen oder auch nach Einträgen suchen. Das geht über das Register [Anzeigen]. Klicken Sie auf diesen Reiter (Abb. 14.8):

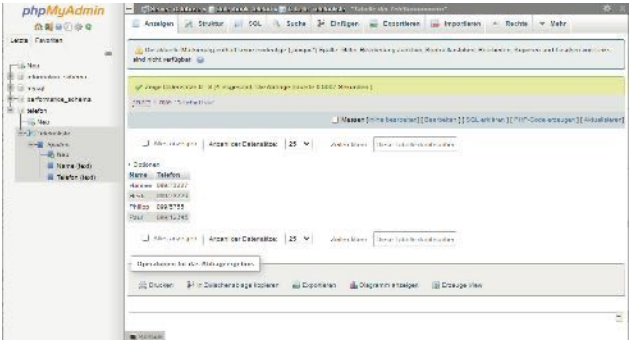



**Abb. 14.8** Die Anzeigenseite von „phpMyAdmin“

In diesem Kapitel haben Sie ein einfaches Beispiel kennengelernt, bei dem eine SQL-Datenbank zusammen mit einem Verwaltungstool für SQL-Datenbanken als Docker-Anwendung aufgesetzt wurde und wie mit dieser Anwendung eine Datenbank verwaltet beziehungsweise bearbeitet werden kann. Es wurde in dieser Datenbank eine Tabelle angelegt und Sie haben die Tabelle mit Daten gefüllt.

Im nächsten Kapitel wollen wir eine einfache Webseite mit PHP-Anweisungen erstellen, welche auf diese Datenbank zugreift und die Daten anzeigt.

## 14.2 Abfrage der Datenbank über PHP

Bis jetzt erfolgte der Zugriff auf die Datenbanken in den Containern über fertige Applikationen wie WordPress oder phpMyAdmin. Sie möchten jetzt aber sicher auch wissen, wie Sie auf die Datenbank, von einem Container mit Ihrer eigenen Anwendung oder aus Ihrer Webseite heraus, zugreifen können.

Zu diesem Zweck erhalten Sie hier ein einfaches Beispiel, wie man über eine Webseite mit PHP auf eine SQL-Datenbank zugreifen kann, die über einen MySQL oder einen MariaDB Container verwaltet wird.



Wir verwenden hier noch einmal die Telefon-Datenbank mit der Telefonliste aus dem vorherigen Kapitel. Um darauf mit unserer eigenen Webanwendung zugreifen zu können, erstellen wir einen weiteren Container aus einem Image, das vom PHP Image aus dem Docker Hub abgeleitet wird.

Das Dockerfile, mit dem wir dieses Image erzeugen, entspricht in etwa dem aus dem Beispiel Telefon-PHP im Kapitel 4.9. Dieses Dockerfile wird aber durch zusätzliche Anweisungen erweitert, welche die benötigten SQL-Erweiterungen für PHP aktivieren.

Wir legen für dieses Image das folgende neue Verzeichnis an:

```
1  '<USER_DIR>\Telefon-DB'
```

Dort wird das Dockerfile angelegt und mit den unten dargestellten Einträgen gefüllt.

```
Datei 'dockerfile'

FROM php:7.2-apache
COPY src/ /var/www/html/
RUN docker-php-ext-install mysqli \
    && docker-php-ext-enable mysqli
```

Das Image wird auch hier auf Basis des Images ‚php:7.2' in der Variante ‚apache' aus dem Docker Hub erstellt.

Das Verzeichnis mit dem Namen ‚src', in dem sich die Datei ‚index.php' befindet soll, wird im Image in das Standard-Verzeichnis. `'/var/www/html/'` kopiert.

Mit der abschließenden RUN-Anweisung werden PHP-Erweiterungen für die im PHP-Code benutzten MySQL-Kommandos installiert und aktiviert.

Jetzt muss noch die Datei `'index.php'` erstellt werden. Dort befinden sich die PHP-Kommandos, mit denen wir auf die Telefon-Datenbank zugreifen. Wir lesen mit PHP die Daten aus der Datenbank aus und zeigen sie im HTML-Format an.

Wir erzeugen wieder ein Unterverzeichnis mit dem Namen `'src'` unterhalb des Verzeichnisses `'Telefon-DB'`.

```
<USER_DIR>\Telefon-DB\src
```

Dort kommt die Datei `'index.php'` hin. Damit wird sie durch die Anweisung im Dockerfile in den Container an die Stelle kopiert, wo sie standardmäßig gesucht und ausgeführt wird.

Übernehmen Sie jetzt den PHP-Code aus der folgenden Vorlage in die Datei.

```
Datei 'index.php'
```

```
2
3  <?php
4
5  define ( 'MYSQL_HOST',      'mariadb_database_1' );
6  define ( 'MYSQL_USER',      'root' );
7  define ( 'MYSQL_PASSWORD',  'topsecret' );
8  define ( 'MYSQL_DATABASE',  'telefon' );
9
10 $db_link = mysqli_connect (
11                      MYSQL_HOST,
12                      MYSQL_USER,
13                      MYSQL_PASSWORD,
14                      MYSQL_DATABASE
15                     );
16
17 $sql = "SELECT * FROM Telefonliste";
18
19 $db_erg = mysqli_query( $db_link, $sql );
20 if ( ! $db_erg )
21 {
22     echo "<td>". "Invalid Query - " . $db_erg . "</td>";
23     exit(1);
24 }
25 echo "<td>". "Query - Success" . "</td>";
26
27 echo '<table border="1">';
28 while ($zeile = mysqli_fetch_array( $db_erg, MYSQLI_ASSOC))
29 {
30   echo "<tr>";
31   echo "<td>". $zeile['Name'] . "</td>";
32   echo "<td>". $zeile[,Telefon'] . "</td>";
33   echo "</tr>";
34 }
35 echo "</table>";
36
37 mysqli_free_result( $db_erg );
38
39 mysqli_close($db_link);
40 ?>
```

Mit den ersten Anweisungen werden Konstanten deklariert. die beim Aufruf der SQL Connect-Funktion als Parameter übergeben werden. Dazu gehört auch die IP-Adresse des Datenbank-Hosts. Wir geben hier den Namen des Containers an. Der wird durch die interne DNS-Funktionalität von Docker durch die passende IP ersetzt. Da wir in unserem Beispiel mit Docker Compose arbeiten, wurde von Docker der Contai-

ner-Name `'mariadb_database_1'` vergeben, gemäß den internen Regeln von Docker zur Definition von Container-Namen (siehe Kapitel 12.8.2).

Wie Sie wahrscheinlich noch wissen, werden mit dem CLI-Kommando `docker container ls` die Namen der aktuell laufenden Container in der letzten Spalte (NAMES) angezeigt und Sie können einen Container-Namen dadurch herausfinden.

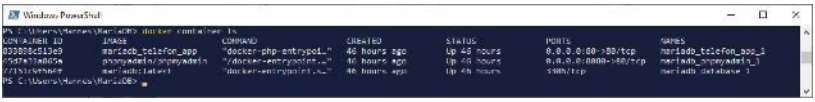



**Abb. 14.9** Anzeige von Container-Namen durch das Kommando `docker container ls`

Die Werte der anderen Konstanten, also der Benutzername, das zugehörige Passwort und der Datenbank Name, müssen mit den Angaben übereinstimmen, die zum Service `database` in der Datei `'dockercompose.yaml'` eingetragen werden.

Es folgt dann ein Connect-Kommando, mit dem die Verbindung zur Datenbank aufgebaut wird. Das Kommando liefert im Erfolgsfall eine Link ID als MySQL-Verbindungskennung zurück. Im Fehlerfall wird FALSE zurückgegeben.

Wir stellen ein SQL-Kommando als String zusammen und übergeben diesen SQL-String an die Variable `$sql`.

Dann wird die eigentliche Abfrage, die SQL-Query, durchgeführt. Als Parameter benötigt die aufgerufene Funktion `mysqli_query()` die Link ID der Datenbank und den Query String. Die Funktion liefert das Ergebnis der Abfrage als Rückgabewert. Der wird in der Variable `$db_erg` zwischengespeichert.

Jetzt wird geprüft, ob die Abfrage erfolgreich war. Wenn nicht, wird das Programm mit einer Fehlermeldung beendet.

War die Abfrage erfolgreich, wird das Ergebnis der Abfrage in einer Schleife ausgewertet und die Daten werden im HTML-Format als Tabelle ausgegeben.

Zuletzt wird aufgeräumt und die Verbindung zum Datenbank-Host wird geschlossen.

Im Verzeichnis

```
<USER_DIR>\MariaDB
```

erweitern wir die Docker-Compose-Datei mit den Einträgen für einen weiteren Container, der den Service `telefon_app` instanziiert und dessen Eigenschaften festlegt.

```
Datei 'docker-compose.yaml'

version: "3.7"
services:

    database:
        image: mariadb:latest

        restart: always

        volumes:
            - mariadb_data:/var/lib/mysql

        environment:
            MYSQL_ROOT_PASSWORD: topsecret
            MYSQL_DATABASE: telefon
            MYSQL_USER: user
            MYSQL_PASSWORD: topsecret

    phpmyadmin:

        image: phpmyadmin/phpmyadmin

        ports:
            - "8080:80"

        restart: always

        environment:
```

```
30              PMA_HOST: database
31
32          depends_on:
33              - database
34
35      telefon_app:
36          build: ../Telefon-DB
37          ports:
38              - "80:80"
39          volumes:
40              - ../Telefon-DB/src/:/var/www/html/
41
42          environment:
43              MYSQL_HOST: database
44
45          depends_on:
46              - database
47
48          volumes:
49      mariadb_data:
```

Wir geben hier an, dass der Service den Namen `telefon_app` erhält und der Container dafür mit dem Dockerfile aus dem Verzeichnis `'Telefon-DB'` erzeugt werden soll.

Der Port 80 aus dem Host wird auf den internen Port 80 im Container gemappt.

Das lokale Host-Verzeichnis ‚`../Telefon-DB/src/`' wird in das Standard-Verzeichnis `'/var/www/html/'` gemapped. Damit können wir die Auswirkung von Änderungen in der Datei `'index.php'` direkt auf dem Browser ansehen, ohne dass der Container neu gebaut werden muss.

Der Environment-Variablen `MYSQL_HOST` wird der Name des Datenbank-Containers zugewiesen. Mit dem Eintrag `depends_on` geben wir bekannt, dass auch dieser Service erst gestartet werden soll, wenn der Start des Datenbank-Containers abgeschlossen ist.

Damit sind wir soweit, dass unsere neue Applikation gestartet werden kann. Dazu starten wir eine Shell, wechseln in das Verzeichnis

```
<USER_DIR>\MariaDB
```

und starten die Services:

```
docker-compose up -d --build
```

Mit dem Schalter `--build` soll sichergestellt werden, dass das Image aus dem Dockerfile im Verzeichnis **<USER_DIR>\Telefon-DB** ganz sicher wieder neu gebaut wird, auch wenn es schon von einem vorherigen Build existieren sollte. Es kann sonst passieren, dass Änderungen im Dockerfile nicht ins Image übernommen werden und somit keine Wirkung haben.

Starten Sie einen Internet-Browser. Als URL geben sie wieder einmal `http://localhost` diesmal mit der Portnummer 80 an (die Portnummer der Telefon-App).

```
http://localhost:80/
```

Jetzt wird eine Webseite angezeigt, die den Inhalt der Telefonliste aus unserer Datenbank in einer einfachen Tabelle ausgibt.

Query - Success

| | |
|---|---|
| Hannes | 089/73227 |
| Heidi | 089/73226 |
| Philipp | 099/5755 |
| Paul | 099/12345 |
| Simon | 0234/6789 |
| Georg | 099/131085 |

Übrigens: Wenn Sie localhost mit der Portnummer 8080 aufrufen, landen Sie immer noch auf der Loginseite von „phpMyAdmin“. Nachdem Sie sich dort angemeldet haben, können sie die Datenbank weiterbearbeiten und auch neue Daten einfügen.

## 14.3 Übungsaufgabe: Die Telefon-App bearbeiten

Passen Sie bei dieser Übung den Code für die „Telefon-App“ in der Datei `'index.php'` so an, dass Ausgabe und Funktion der Webseite dem Beispiel von „Telefon-PHP“ aus Kapitel 4.9 entsprechen. Die Telefon-Daten sollen jetzt allerdings nicht mehr aus einer statischen Tabelle im PHP-Code gewonnen werden, sondern aus der SQL-Datenbank des Datenbank-Containers.

***Lösung:***

Ein Beispiel zur Lösung die Datei ‚index.php‘:

```
Datei 'index.php'
<!DOCTYPE html>
<html lang="de">
<head>
      <title> Telefon Liste PHP </title>
</head>

<body bgcolor="yellow">
<h1>Telefonnummer Suche</h1>

<!-- Eingebeddeter PHP Code -->
<?php
if(!isset($_GET['surname']))
{
      $currentName= "";
}
else
{
      $currentName = $_GET['surname'];
}

if($currentName != "")
{
      if(checkName($currentName))
      {
            echo "Der Name ". $currentName . " wurde gefunden";
      }
      else
      {
            echo "Unbekannter Name: ". $currentName;
      }
}

function checkName($userName)
{
    $retval = false;

    define ( 'MYSQL_HOST',      'mariadb_database_1' );
    define ( 'MYSQL_USER',      'root' );
    define ( 'MYSQL_PASSWORD',  'topsecret' );
    define ( 'MYSQL_DATABASE',  'telefon' );

    $db_link = mysqli_connect (
```

```
                        MYSQL_HOST,
                        MYSQL_USER,
                        MYSQL_PASSWORD,
                        MYSQL_DATABASE
                       );

    $sql = "SELECT * FROM Telefonliste WHERE Name =
    '$userName'";

    $db_erg = mysqli_query( $db_link, $sql );

    if ( $db_erg )
    {
        $zeile = mysqli_fetch_array( $db_erg, MYSQLI_ASSOC);
        $tel = $zeile['Telefon'];
        if($tel != "")
        {
            echo "<p>" . "Name: $userName - „ . "Telefon: " .
            $tel . "</p>";
            $retval = true;
        }
    }
    mysqli_free_result( $db_erg );

    mysqli_close($db_link);

      return $retval;
}
?>

<!-- Formular Bereich -->
<form action="index.php" method="get">

<p>Geben Sie einen Namen ein:
<input type="text" name="surname">
</p>
<p>
<input type="submit" value="Suchen">
</p>
</form>

</body>
```

Die Anzeige im Webbrowser unterscheidet sich nicht von der aus dem Beispiel von Kapitel 4.9 (siehe Abb. 4.31). Die Telefondaten werden hier aber im Hintergurnd aus der Datenbank gewonnen.

Kapitel 15

# Docker Swarm

## 15.1 Was ist Docker Swarm

Als Einstieg wollen wir die Frage klären, was das überhaupt sein soll, ein Docker Swarm beziehungsweise was der Swarm Mode bedeutet.

Wenn unter Docker mehrere physikalische oder virtuelle Maschinen zu einer Gruppe zusammengefasst werden sollen, um gemeinsam, in einem sogenannten Cluster, eine Docker-Applikation zu realisieren, dann kann so etwas mithilfe von Docker Swarm organisiert werden.

Docker Swarm übernimmt dabei die Aufgabe eines Orchestrierungs-Tools. Diese Art von Tools unterstützen uns dabei, Systeme zu verwalten, die aus zahlreichen Containern bestehen, welche wiederum über viele Host-Rechner verteilt sein können.

Der wichtigste Vorteil eines Systems, das im Swarm Mode betrieben wird, besteht darin, dass ein hohes Maß an Ausfallsicherheit und Verfügbarkeit gewährleistet werden kann. Ein Docker Swarm besteht in der Regel aus mehreren Worker (Arbeiter-)Nodes, welche die eigentliche Funktionalität zur Verfügung stellen. Organisiert und koordiniert werden die Worker Nodes dann von mindestens einem oder eben auch mehreren Manager Nodes, welche sicherstellen, dass die verfügbaren Systemressourcen effektiv und zuverlässig eingesetzt werden.

Eine stetig wachsende Zahl von Entwicklern übernimmt Docker Container-Technologien zusammen mit Docker Swarm, um die Entwicklung von Applikationen effektiver zu gestalten und deren Betriebssicherheit zu erhöhen.

Im Swarm-Modus können zahlreiche Container gleichzeitig zur Bearbeitung einer Aufgabe bereitgestellt werden. Die Skalierung, damit ist die Anzahl von Instanzen eines Containertyps gemeint, kann je nach Systemlast erhöht oder verringert werden. Bei Fehlschlag können Container automatisch wieder gestartet oder sogar ersetzt werden.

Docker Swarm bietet darüber hinaus eine automatisierte Verteilung der Last auf die vorhandenen Ressourcen, das sogenannte „Load Balancing". Damit wird sichergestellt, dass bei zahlreichen Anfragen an eine Applikation alle betroffenen Container gleichmäßig ausgelastet werden.

Docker Swarm ist mittlerweile (seit Docker Version 1.12) integraler Bestandteil von Docker und muss bei Bedarf eigentlich nur noch aktiviert werden. Deshalb spricht man heutzutage auch vom Swarm Mode.

Im Swarm Mode wird automatische Sicherheit durch TLS-Verschlüsselung (das Transport Layer Security-Protokoll) und die gegenseitige Authentifizierung der Nodes bereitgestellt. Man hat hier die Möglichkeit, selbstgezeichnete Root-Zertifikate zu nutzen oder von einer Zertifizierungsstelle ausgestellte Zertifikate.

## 15.2 Neue Begriffe für den Swarm Mode



Cluster: Ein Verbund von virtuellen und physikalischen Maschinen zur Steigerung von Rechenleistung und zur Verbesserung der Ausfallsicherheit.

Orchestrierung: Die flexible und automatisierte Kombination, Konfiguration und Koordination verschiedener Computer und deren Dienste zu einem Gesamtsystem.

Node: Ein Node (oder auch Knoten) ist eine Instanz einer Docker Engine, auf welcher der Swarm Mode aktiviert ist und die zu einem Docker Swarm gehört. Wir unterscheiden zwei Arten von Nodes, nämlich Manager Nodes und Worker Nodes.

Manager Nodes: Manager Nodes sind dafür verantwortlich, dass die Aufgaben, welche als Tasks bezeichnet werden, den Worker Nodes zugeteilt werden. Manager Nodes sind darüber hinaus für die Orchestrierung und das Management eines Clusters verantwortlich. Sie müssen in diesem Zusammenhang sicherstellen, dass der festgelegte Status eines Swarm immer aufrechterhalten wird.

Worker Nodes: Diese Nodes führen die Tasks aus, welche ihnen von den Manager Nodes zugeteilt worden sind. Standardmäßig können auch Manager Nodes zusätzlich die Funktionalität von Worker Nodes übernehmen. Ein Worker Node informiert seinen Manager Node ständig über seinen aktuellen Zustand. Damit ist ein Manager Node in der Lage, den definierten Status des Clusters und dessen Nodes aufrecht zu erhalten.

Task: Ein Task ist die kleinste ausführbare Einheit eines Docker Swarm. Ein Task beinhaltet die Instanz eines Docker Containers mit den Kommandos, welche die servicespezifischen Aufgaben erledigen. Manager Nodes weisen Tasks den Worker Nodes in einer festgelegten Anzahl von Instanzen zu.

Service: Die Kombination der Tasks, welche auf einem Manager oder Worker Node laufen, nennt man auch Service. Dabei wird spezifiziert, aus welchem Container Image die Tasks aufgebaut werden und welche Kommandos zur Laufzeit innerhalb eines Containers ausgeführt werden.

Replicated Service: Ein Docker Service, bei dem eine bestimmte Anzahl von Kopien läuft. Diese Kopien sind allesamt Instanzen eines Docker Containers.

Global Service: Docker Swarm führt einen Task des Service in jedem Node eines Clusters aus.

Load Balancing: Beim Load Balancing geht es um die gleichmäßige Verteilung der Gesamtlast auf parallel arbeitende Tasks. Damit soll die Effektivität der Verarbeitung insgesamt verbessert werden. Für diese Aufgabe wurden zahlreiche Algorithmen entwickelt, die zum Teil sehr komplex sind. Ein Swarm Manager benutzt das interne Load Balancing, um Anfragen auf die Services eines Clusters zu verteilen.

Skalierung: Für jeden Service kann die Anzahl der Tasks bestimmt werden, die für diesen Service ausgeführt werden.

Secrets: In der Docker-Welt spricht man von „Secrets" (Geheimnissen), wenn es sich um schützenswerte oder sicherheitskritische Daten handelt, wie zum Beispiel Passwörter, Schlüssel, Zertifikate oder Ähnliches. Vor allem wenn diese Informationen über das Netzwerk übertragen oder lokal gespeichert werden sollen, bietet Docker im Swarm Mode einen speziellen Mechanismus dafür an.



Certificates: Bei zertifikatbasierter Authentifizierung wird ein digitales Zertifikat (Certificate) verwendet, um eine Entität (einen Benutzer, ein Gerät oder ein System) zu identifizieren, bevor der Zugriff auf eine Ressource, ein Netzwerk oder eine Anwendung gewährt wird. Zertifikate werden unter anderem von speziell zugelassenen Zertifizierungsstellen ausgestellt. Diese werden auch als CA (Certificate Authority) bezeichnet.

Stack: Mit Docker Stack ist es möglich, mehrere Docker Services zu Multi-Container-Applikationen zu verknüpfen. Um dies zu konfigurieren, wird für Docker Stack eine Konfigurationsdatei im YAML-Format wie bei Docker Compose erstellt. Dort gibt es zusätzliche Einträge, die das Deployment der Services spezifizieren.

Raft Datenbank: Im Swarm-Modus wird von allen Manager Nodes eine eigene Instanz einer Datenbank integriert, durch welche der globale Status aller Cluster verwaltet wird. Dieses Datenbank-System arbeitet mit dem sogenannten „Raft Distributed Consensus Algorithmus". Dieser Algorithmus stellt sicher, dass der Inhalt aller Datenbankinstanzen, auf die in den Manager Nodes zugegriffen wird, konsistent ist. Vereinfacht gesagt wird sichergestellt, dass in den Datenbanken aller Cluster die gleichen Daten vorhanden sind.

## 15.3 Einen Single Node Swarm erstellen

Da in diesem Buch die Beispiele und Erklärungen zu Swarm für den Swarm Mode konzipiert sind, muss sichergestellt werden, dass die Client-API und die Daemon-API von Docker in der Version 1.24 oder höher vorliegen.

Überprüfen können Sie dies mit dem folgenden CLI-Kommando:

```
1 > docker version
```

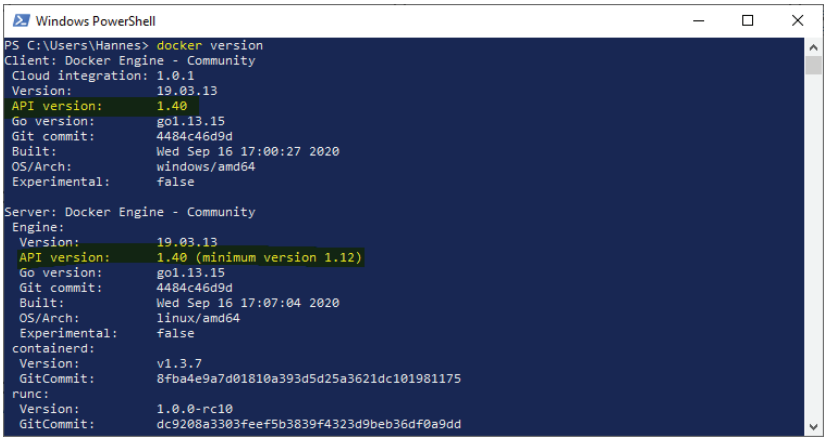



**Abb. 15.1** Abfrage der Docker-Version

### 15.3.1 Initialisierung des Docker Swarm Modes

Wie wir bis jetzt schon gelernt haben, ist Docker Swarm eine Technologie, welche die Verteilung von Aufgaben in Clustern ermöglicht, deren Nodes auf verschiedene Host-Rechner verteilt sein können.

Um den Einstieg in den Swarm Mode von Docker zu erleichtern, beginnen wir aber zunächst mit einem Single Node Swarm. Das heißt, in unserem Schwarm gibt es nur einen „Fisch" – es ist also nicht wirklich ein Schwarm. Diese Konfiguration erlaubt es uns aber trotzdem, die ersten Kommandos für Docker Swarm auszuprobieren. Damit können wir uns an die Arbeit mit Docker Swarm gewöhnen, obwohl wir nur eine Docker Engine auf einem Host-Rechner am Laufen haben.

Bevor wir uns in den späteren Kapiteln in die Multi-Node-Kommandos einarbeiten, müssen wir dafür dann allerdings doch eine Multi-Node-Arbeits- und Testumgebung mit mehreren Hosts schaffen. Das müssen aber nicht unbedingt physikalische Rechner sein, sondern wir können dazu auch virtuelle Maschinen aufsetzen.

Nach der Installation ist der Swarm Mode standardmäßig nicht aktiviert. Damit wir Swarm-Kommandos ausführen können, müssen wir den Swarm Mode deshalb initialisieren.

Starten Sie dazu eine Shell und geben Sie das folgende Kommando ein (Abb. 15.2):

```
> docker swarm init
```

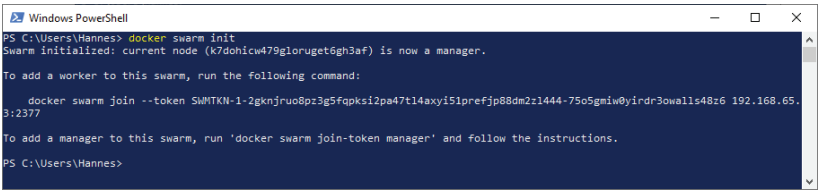



**Abb. 15.2** Initialisieren des Docker Swarm Modes

War die Initialisierung erfolgreich, so teilt uns Docker dies mit. Es wird bei einer Single Node-Konfiguration der aktuelle Node automatisch als Manager Node konfiguriert. Das wird in unserem Beispiel auch zusammen mit der Node ID zurückgemeldet.

Docker ist auch so freundlich, uns als Ausgabe eine Kopiervorlage anzuzeigen, mit der wir als Nächstes dem Manager Node einen Worker Node zufügen könnten.

In unserem Beispiel würde das Kommando wie folgt aussehen:

```
> docker swarm join --token SWMTKN-1-2gknjruo8pz3g5fqpksi2pa47tl
4axyi51prefjp88dm2zl444-75o5gmiw0yirdr3owalls48z6
192.168.65.3:2377
```

Wenn wir uns später eine Multi-Node-Umgebung eingerichtet haben, dann werden wir das Kommando wieder aufgreifen und praktisch ausprobieren.

Damit haben wir auch schon unsere ersten zwei Kommandos für den Swarm Mode kennengelernt.

Hier die Syntax für das Init-Kommando:

```
1 > docker swarm init [OPTIONS]
```

An dieser Stelle wollen wir nur eine Option vorstellen, die Option `--advertise-addr`. Mit diesem Flag gibt man die Adresse an, die für den API-Zugriff über das Netzwerk an andere Mitglieder des Swarms bekannt gegeben wird. Wird diese Angabe weggelassen, so überprüft Docker, ob es für das zugehörige System eine einzige IP-Adresse gibt, und nutzt diese auf dem Listening-Port. Besitzt das aktuelle System mehrere IP-Adressen, so muss diese Option unbedingt mit angegeben werden.

Sehen wir uns auch die Syntax für das Join-Kommando genauer an:

```
1 > docker swarm join [OPTIONS] <HOST:PORT>
```

Im Moment interessiert uns auch hier nur die eine Option, die ja auch bei der Ausführung des Init-Kommandos in der Ausgabe mit vorgeschlagen wurde, die Option `--token`. Damit gibt man einen sogenannten Join-Token an, der für den Zutritt zu einem Swarm benötigt wird.

Damit die Worker den zugeordneten Manager erreichen können, benötigen sie dessen IP-Adresse und die Portnummer. Diese Informationen stehen am Ende des CLI Join-Kommandos.



Im Moment können wir das Kommando noch nicht praktisch ausprobieren, da wir ja zuerst einmal nur mit einem einfachen ‚Single Node' Swarm arbeiten wollen. Das wird selbstverständlich in einem späteren Kapitel dieses Buches nachgeholt.

Falls Sie irgendwann einmal nicht sicher sind, ob der Swarm Mode aktiv ist oder nicht, dann können Sie sich diese Information mithilfe des `docker info`-Kommandos an dem zugehörigen Node ausgeben lassen:

```
1 > docker info [OPTIONS]
```

Als einzige Option gibt es dabei das Format Flag `--format` oder `-f`. Das kennen Sie bereits von anderen Kommandos. Man übergibt damit einen Format-String im Go-Format. Die Anwendung dieses Formats wird im Kapitel *19.11 Format Angaben für Docker-Kommandos* beschrieben.

Nach der Eingabe des Kommandos `docker info` erhalten Sie die Ausgabe mit den aktuellen, systemweiten Docker-Informationen, darunter auch den Swarm Mode (Abb. 15.3):

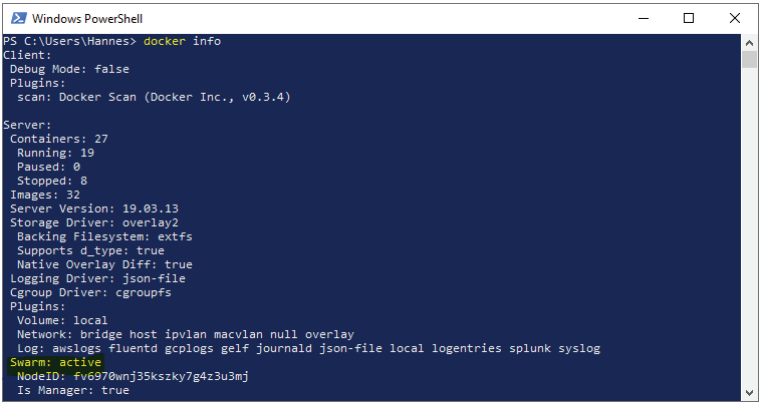



**Abb. 15.3** Docker-Info: Swarm Mode abfragen

Im Screenshot wird angezeigt, dass der Swarm Mode „active“ ist. Ansonsten stünde hier „inactive“.

### 15.3.2 Docker-Kommandos zur Node-Verwaltung

Nachdem Docker Swarm auf unserem Host-Rechner initialisiert worden ist, hat Docker in diesem Zusammenhang unser System als Knoten vom Typ Manager dem Swarm hinzugefügt.

Damit haben wir jetzt die Möglichkeit, einige Kommandos kennenzulernen, die uns Docker für die Verwaltung von Swarm Nodes zur Verfügung stellt.

Die Syntax des übergeordneten Node-Kommandos sieht so aus:

```
1  > docker node <COMMAND>
```

Lassen wir uns als Erstes eine Liste aller Nodes im Swarm anzeigen. Das Kommando dazu lautet (Abb. 15.4):

```
1  > docker node ls
```


**Abb. 15.4** Docker Node: Anzeige einer Node-Liste

Der Screenshot zeigt uns, dass es in diesem Swarm nur einen Node gibt. Dabei wird die Node ID, der Name des Hosts, auf dem der Node ausgeführt wird, der Node-Status, die Verfügbarkeit des Nodes, der Manager-Status und die Version der zugehörigen Docker Engine ausgegeben.

In der Spalte „Availability" wird informiert, ob einem Node vom Scheduler Tasks zugewiesen werden können oder nicht. „Active" bedeutet, dass Tasks zugewiesen werden können. „Pause" gibt an, dass vom Scheduler keine neuen Tasks zugewiesen werden sollen, vorhandene laufen aber weiter. „Drain" gibt an, dass vom Scheduler keine neuen Tasks zugewiesen werden sollen. Der Scheduler fährt dabei vorhandene Tasks herunter und weist sie anderen verfügbaren Nodes zu.

Der Manager-Status des angezeigten Knotens ist in dem gezeigten Beispiel Leader. In einem Cluster können ja mehrere Manager Nodes zusammenarbeiten. Nur einer von diesen kann aber der „Chef" sein. Dieser bekommt den Status „Leader".

„Leader" bedeutet für einen Manager Node, dass dieser der führende Manager Node eines Clusters ist. Dieser trifft alle Entscheidungen, welche die Verwaltung und Orchestrierung des zugehörigen Swarms angehen.

Neben dem Status „Leader" gibt es noch „Reachable", der Node ist erreichbar oder „Unavailable", also nicht erreichbar. Bei Nodes ohne Statusangabe handelt es sich um Worker Nodes, die keine Management-Funktion ausüben.

Wenn in einem Cluster mit mehreren Manager Nodes derjenige mit dem Status „Leader“ ausfällt, wird automatisch ein anderer Manager Node zum „Leader“ befördert.

Wollen wir genauere Informationen über einen Node erhalten, gibt es dafür den „Inspect“-Befehl. Hier die Syntax:

```
> docker node inspect [OPTIONS] self|<NODE> [NODE...]
```

Als Option ist hier auch die Angabe `--format` oder `-f` möglich, um eine formatierte Ausgabe im Go-Format zu erhalten. Es gibt auch noch das Flag `--pretty` als Option, welches eine Anzeige in einem besser lesbaren Format bewirkt.

Möchten Sie Informationen über den eigenen Knoten erhalten, dann genügt die Angabe `"self"` zur Identifizierung des gewünschten Nodes. Für die anderen Nodes in einem Cluster muss die Node ID mit angegeben werden. Es müssen aber nicht alle Zeichen der ID eingetippt werden. Es genügen die ersten paar Zeichen der ID, aber es müssen ausreichend viele Zeichen sein, um einen Knoten eindeutig zu identifizieren.

Hier ein Screenshot mit einem Beispiel-Kommando (Abb. 15.5):

```
> docker node inspect --pretty k7do
```

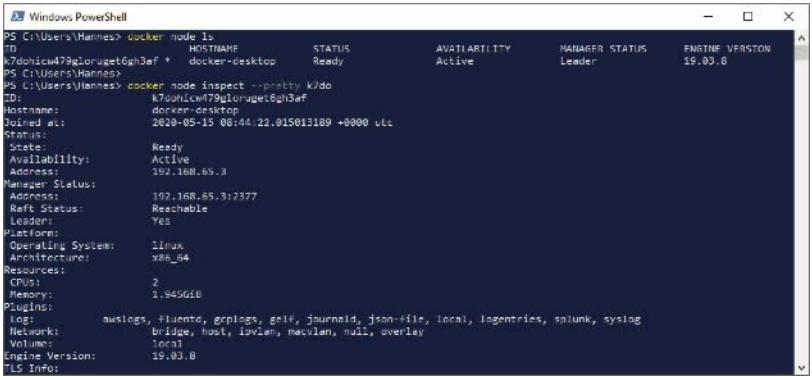



**Abb. 15.5** Docker Node Inspect: Anzeige von Node-Informationen

Die nächsten Node-Kommandos können wir noch nicht sinnvoll ausprobieren. Das folgt später, wenn unsere Cluster mehr Funktionalität erhalten haben. Wir führen sie hier aber mit auf, um eine vollständige Übersicht über die verfügbaren Node-Kommandos von Docker zu geben.

Ausgabe der Task in einem Node:

```
> docker node ps [OPTIONS] [NODE...]
```

Entfernen eines Nodes aus einem Swarm:

```
> docker node rm [OPTIONS] [NODE...]
```

Beförderung eines Nodes zum Manager (kann nur von einem Manager Node ausgeführt werden):

```
> docker node promote <NODE>
```

Degradierung eines Manager Nodes (kann nur von einem Manager Node ausgeführt werden):

```
> docker node demote <NODE>
```

## 15.4 Docker Services

### 15.4.1 Einen Service erstellen



Zur Wiederholung: Bei einem Service handelt es sich um einen Container oder mehrere Container mit der gleichen Konfiguration, die im Swarm Mode von Docker ausgeführt werden.

Unser einfacher Single Node Swarm hat im Moment noch keine Funktion. Er kann ja auch nur managen und es gibt in unserem Schwarm noch nichts zu verwalten. Dazu brauchen wir einen Service, der dem Swarm dann zugeteilt wird.

Das Docker-Kommando, mit dem man einen Service erstellen kann, lautet `docker service create`. Hier die Syntax:

```
1  > docker service create [OPTIONS] <IMAGE> [COMMAND] [ARG...]
```

Da es sich bei diesem Kommando um ein Cluster Management-Kommando handelt, funktioniert es nur, wenn es auf einem Swarm Manager Knoten ausgeführt wird.

Für dieses Kommando steht eine umfangreiche Liste von Optionen zur Verfügung. Diese lassen wir aber in unserem ersten Beispiel erst einmal außen vor und geben nur den Namen eines Docker Images an (Abb. 15.6).

```
1  > docker service create redis
```

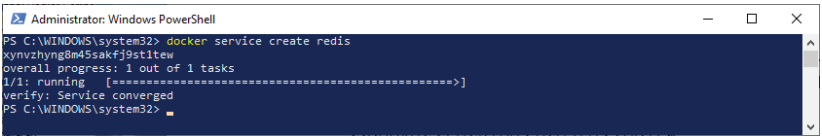



**Abb. 15.6** Docker Service Create: einen einfachen Service erstellen

Mit diesem Kommando erstellen wir einen Service mit einem einzigen Task. Dieser Task führt einen Container aus, der das „redis“ Image aus dem Docker Hub instanziiert.

Das Erstellen eines Service nimmt, je nach Image, in der Regel einige Zeit in Anspruch. Bitte haben Sie etwas Geduld und brechen Sie den Vorgang nicht vorzeitig ab.

Bei Redis handelt es sich um ein Open-Source-Produkt, das einen In-Memory-Key-Value-Datenspeicher realisiert. Da wir Redis nur als Übungsbeispiel verwenden wollen und daher auch keine weiteren Funktion nutzen, lassen wir hier zusätzliche Parameter weg, die normalerweise für ein Image beim Start eines Service mit angegeben werden.

Jetzt können wir wieder einige neue Kommandos für Docker Services testen.

### 15.4.2 Eine Liste der Services ausgeben

Das Kommando `docker service` kennt ebenfalls die Option `ls`, mit welcher eine Liste der Services in einem Swarm ausgegeben wird. Probieren wir es aus (Abb. 15.7):

```
1  > docker service ls
```

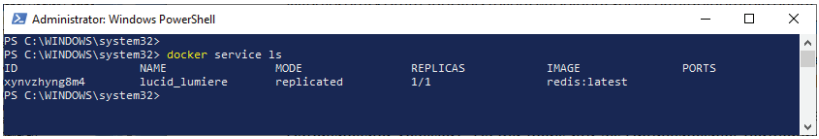



**Abb. 15.7** Docker Service ls: eine Liste der Services in einem Node ausgeben

Auch dieses Kommando ist ein Cluster Management-Kommando und kann nur auf einem Swarm Manager-Knoten ausgeführt werden.

Wie wir anhand der Ausgabe sehen können, gibt es in unserem aktuellen Knoten nur einen Service. Dieser hat die ID `xynvzhyng8m4` und den Namen `lucid_lumiere`. Dieser Name wurde von Docker selbst vergeben, da wir beim Erstellen des Service keinen Namen angegeben haben.

Es gibt auch nur eine Instanz des Service. Da wir beim Erstellen keine Angaben über die Anzahl der Replikate gemacht haben, wird dies als Standardwert angenommen.

Es sind auch keine Ports definiert. Auch das hätten wir beim Aufruf von docker service create als Option angeben können, ähnlich wie beim Kommandos `docker run` mit der Publish-Option (`-p` oder `--publish`).

### 15.4.3 Auflistung der Service Tasks

Wir wollen jetzt das Kommando kennenlernen, mit dem man den Status aller Tasks für bestimmte Services abfragen kann.

Docker stellt für das Kommando `docker service` die Option `ps` bereit. Es kann dabei der Prozess-Status von mehreren Services mit einem Kommando abgerufen werden.

```
1 > docker service ps [OPTIONS] <SERVICE> [SERVICE ...]
```

Der Service, für den wir die Tasks anzeigen wollen, wird entweder durch seinen Namen oder durch die ID identifiziert. Auch hier gilt: Bei Angabe der ID genügen die ersten Zeichen, durch welche ein Service von Docker eindeutig identifiziert werden kann.

Für den aktuellen Service gibt es demnach zwei Varianten, um die Tasks aufzulisten:

Mit der Service ID

```
1 > docker service ps xynv
```

oder mit dem Service-Namen

```
1 > docker service ps lucid_lumiere
```

Hier als Beispiel ein Screenshot mit der Service ID (Abb. 15.8).

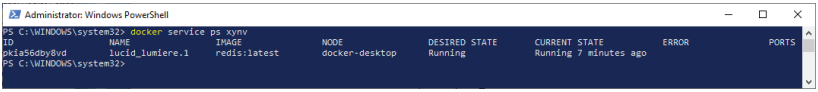



**Abb. 15.8** Docker Service ps: Auflistung der Tasks für einen Service

### 15.4.4 Einen Service entfernen

Wir löschen den gerade erstellten Service wieder. Später, in unserer Übungsaufgabe, starten Sie neue Services mit von Ihnen definierten Namen und mit mehreren Replikaten.

Auch beim Service-Kommando gibt es zum Löschen die Option `rm`:

```
1 > docker service rm <SERVICE>
```

Wieder wird beim Löschen der zu löschende Service entweder durch seinen Namen oder durch die ID identifiziert. Und auch hier gilt bei Angabe der ID: Es genügen die ersten Zeichen, um einen Service für Docker zu identifizieren.

Um den aktuellen Service aus dem obigen Beispiel zu entfernen, gibt es also wieder zwei Varianten:

```
1  > docker service rm xynv
```

oder

```
1  > docker service rm lucid_lumiere
```

Hier als Beispiel ein Screenshot mit dem Service-Namen. Im Anschluss folgt noch einmal das Service-Kommando mit der Option `ls`, um zu zeigen, dass kein Service mehr aktiv ist (Abb. 15.9).

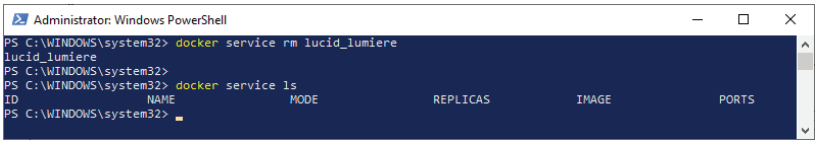



**Abb. 15.9** Docker Service rm: einen Service entfernen



### 15.4.5 Weitere Parameter zum Erzeugen eines Service

An dieser Stelle lernen Sie noch zwei neue Parameter für das Kommando `docker service create` kennen.

Wir wollen jetzt den Namen für unseren Service selbst bestimmen. Es gibt dazu die Option `--name`. Die wird so eingesetzt, wie Sie das von anderen Docker-Kommandos auch schon kennen. Hier die Erweiterung der Syntax dazu:

```
1  > docker service create [--name <NAME>] <IMAGE>
```

Wollen Sie beim Start eines Service die Anzahl der Replikate definieren, so geht das über die Option `--replicas`.

```
1  > docker service create [--replicas=<NUMBER>] <IMAGE>
```

### 15.4.6 Übungsaufgabe: Services mit Replikaten

Als Übungsaufgabe sollen Sie für den aktuellen Node zwei Services anlegen.

Einer soll vier Container-Instanzen aus dem Image redis starten und den Namen `'my_redis'` tragen.

Der andere Service soll `'my_nginx'` heißen und vier Container-Instanzen von nginx als Tasks ausführen.

Lassen Sie sich dann eine Liste der Services anzeigen.

Zuletzt sollen Sie für jeden Service die Liste der Tasks anzeigen, um deren Status zu überprüfen.

***Lösung:***

Starten des Service `my_redis`:

```
1 > docker service create --name my_redis --replicas=4 redis
```

Starten des Service `my_nginx`:

```
1 > docker service create --name my_nginx --replicas=4 nginx
```

Die Liste der Services ausgeben:

```
1 > docker service ls
```

Die Liste der Tasks von `my_redis` ausgeben:

```
1 > docker service ps my_redis
```

Die Liste der Tasks von `my_nginx` ausgeben:

```
1 > docker service ps my_nginx
```

Hier ein Screenshot für dieses Übungsbeispiel aus der PowerShell (Abb. 15.10):

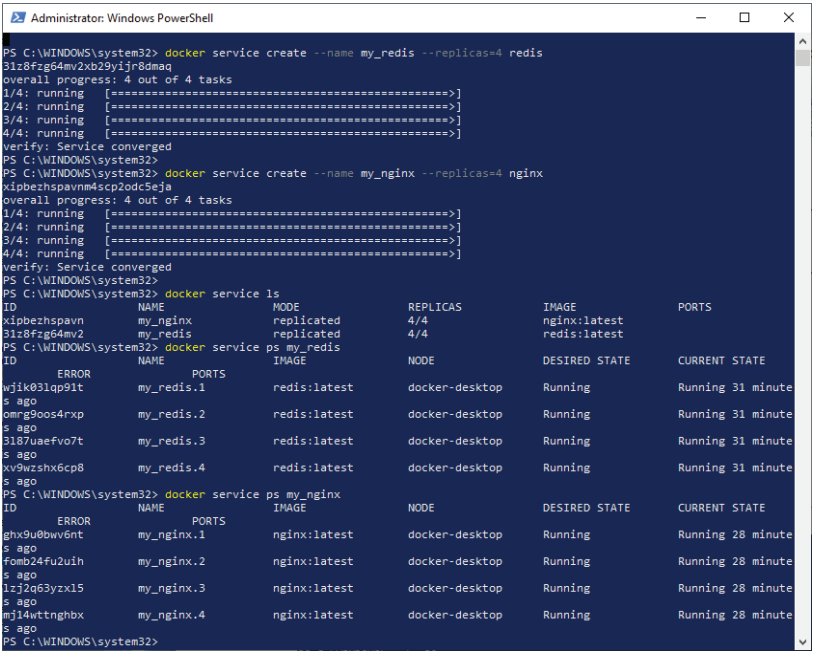



**Abb. 15.10** Übungsaufgabe ‚Docker Services mit Replikaten'

### 15.4.7 Aktualisierung von Docker Services

Das Docker-Kommando `docker service update` verwendet die gleichen Parameter wie das Kommando `docker service create`, um damit die Eigenschaften von laufenden Services zu verändern.

```
1 > docker service update [OPTIONS] <SERVICE>
```

Damit lässt sich zum Beispiel nach dem Service-Start die Anzahl der Replikate eines laufenden Service verändern. Im folgenden Beispiel werden die Replikate auf 6 erhöht (Abb. 15.11):

```
1 > docker service update --replicas=6 my_redis
```

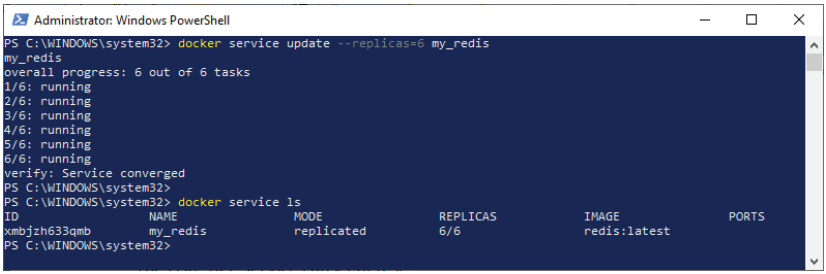



**Abb. 15.11** Docker Service update: einen Service aktualisieren

Hier noch ein Beispiel, mit dem wir bei unserem laufenden Service `'my_redis'` die Anzahl der Instanzen auf 3 verringern und nachträglich einen Service-Port hinzufügen (Abb. 15.12):

```
> docker service update --replicas=3 '
--publish-add published=8080,target=80 my_redis
```

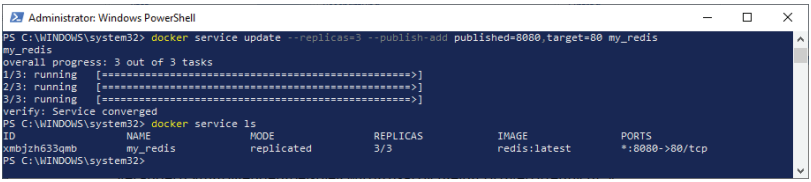



**Abb. 15.12** Docker Service update: Anzahl der Instanzen verändern und einen Port veröffentlichen

Weitere Optionen, die für Services angewendet werden können, wie zum Beispiel Secrets hinzufügen oder Secrets entfernen, behandeln wir später in diesem Buch in eigenen Kapiteln.

### 15.4.8 Docker Services skalieren

Wenn Sie einen laufenden Service nur neu skalieren wollen, also nur die Anzahl der Replikate für einen laufenden Service verändern, dann gibt es für diese Aufgabe neben dem Update-Kommando auch noch ein eigenes Kommando, nämlich das Scale-Kommando.

```
1 > docker service scale <SERVICE=REPLICAS> [SERVICE=REPLICAS...]
```

Dazu ein Beispiel mit einem Kommando, das die Anzahl der Replikate für unseren Service `my_redis` auf 4 Instanzen erhöht (Abb. 15.13):

```
1 > docker service scale my_redis=4
```

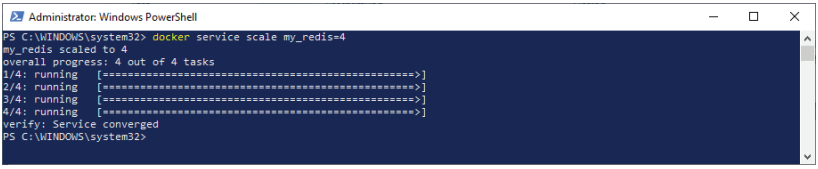



**Abb. 15.13** Docker Service scale: einen laufenden Service skalieren

### 15.4.9 Änderungen an Services rückgängig machen

Falls Sie einmal ein Service Update mit vielen Änderungen durchgeführt haben, und es ist dabei etwas schief gegangen, lassen sich die Änderungen sehr einfach wieder rückgängig machen. Docker bietet dafür das `docker service rollback`-Kommando an. Es ist damit nicht nötig, alle geänderten Werte in einem weiteren Update-Kommando wieder auf den vorherigen Wert zurückzusetzen. Hier wieder die Syntax:

```
1 > docker service rollback [OPTIONS] SERVICE
```

Im nächsten Screenshot sehen Sie ein Beispiel für den Einsatz von Rollback-Kommandos (Abb. 15.14).

Zuerst wird beim Service `my_redis` die Anzahl der Instanzen von 3 auf 4 erhöht.

Mit dem `ls`-Kommando sehen wir uns das Ergebnis des Kommandos an.

Danach machen wir die Änderung mit dem folgenden Kommando wieder rückgängig:

```
1  > docker service rollback my_redis
```

Noch einmal geben wir das `ls`-Kommando ein, um den geänderten Wert für die Anzahl der Replikate zu kontrollieren.

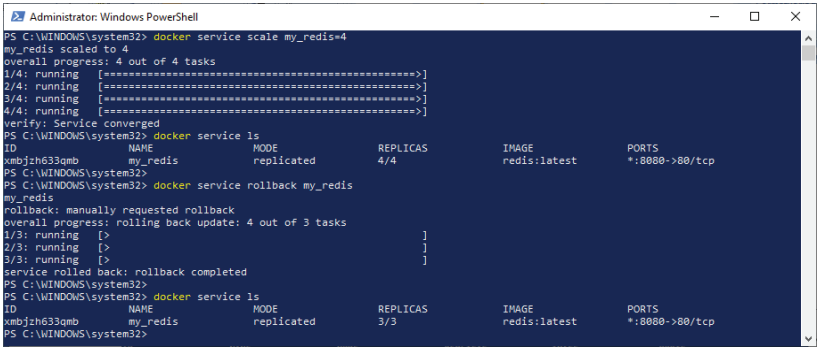



**Abb. 15.14** Docker Service rollback: Service-Änderungen rückgängig machen

### 15.4.10 Ausgabe von Service Logs

Hier noch das Kommando, um die Logs von einem Service zu erhalten:

```
1  > docker service logs [OPTIONS] SERVICE|TASK
```

Die Optionen sind im Großen und Ganzen wie beim `docker log`-Kommando (es sind noch ein paar dazugekommen). Sie können wahlweise angeben, ob sie alle Logs eines Service sehen wollen oder nur die Logs eines Tasks, je nachdem ob Sie den Namen oder die ID des Service oder die ID eines Tasks angeben.

Im Beispiel Screenshot wird zunächst das ps-Kommando für den Service `'my_redis'` ausgeführt. Damit kann aus der ersten Spalte der Ausgabe die ID der zugehörigen Tasks ermittelt werden. Die Task ID aus der ersten Zeile der Liste wird danach für die Ausgabe des zugehörigen Task Logs verwendet (Abb. 15.15).

```
> docker service ps my_redis
> docker service logs xis
```

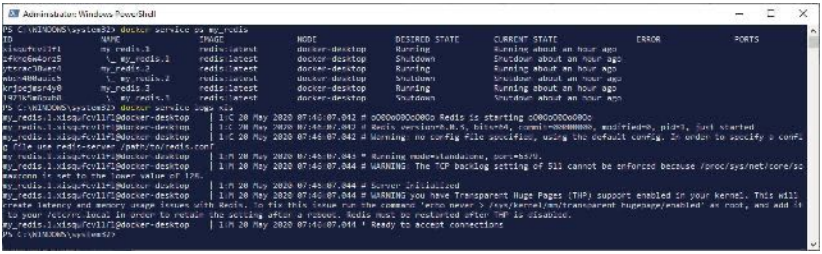


**Abb. 15.15** Docker Servicelogs: Beispiel der Log-Ausgaben für einen Task

### 15.4.11 Ausgabe von detaillierten Service-Informationen

Ausführliche Informationen zu einem Service gibt es durch ein Inspect-Kommando:

```
> docker service inspect [OPTIONS] SERVICE [SERVICE...]
```

Da die Ausgabe des Kommandos sehr umfangreich ist, verwenden wir als Option im folgenden Beispiel das Format-Flag zum Filtern (Abb. 15.16):

```
> docker service inspect '
--format='{{json .Spec.RollbackConfig}}' my_redis
```

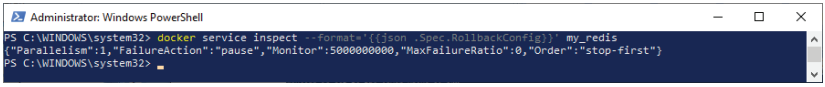



**Abb. 15.16** Docker Service inspect: Beispiel für detaillierte Service-Informationen (mit Format-Flag)

## 15.5 Multi Node Swarm

Jetzt wird es Zeit, dass wir Docker-Applikationen auf mehrere Nodes verteilen, um die Vorteile des Docker Swarm Modes richtig zu nutzen.

Um ein Cluster auf mehrere physikalische Nodes aufteilen zu können, benötigt man mehrere Host-Rechner, auf denen Docker installiert ist. Dort wird auf einem System der Swarm Mode initialisiert. Dabei wird automatisch ein Manager Node mit dem Manager-Status *Leader* eingerichtet.

Auf den anderen Rechnern können dann die Nodes mit dem Docker-Kommando `docker swarm join` angebunden werden. Die Information für das Kommando zum Anbinden eines Worker Nodes mit den benötigten Parametern wird bei der Initialisierung des Manager Nodes durch das Kommando `docker swarm init` als Informationstext ausgegeben.

Vermutlich haben aber die wenigsten Leser zu Hause Zugriff auf einen Serverpark. Darum wollen wir hier erst einmal vorstellen, wie Sie mehrere virtuelle Maschinen auf einem Computer erstellen und dort dann die Nodes Ihres Clusters einrichten und verwalten können.

Eine andere Möglichkeit ist die Installation eines virtuellen Systems wie zum Beispiel VirtualBox oder VMWare auf einem Rechner. Unter Windows 10 mit Hyper-V können mittlerweile mithilfe des HyperV Managers auf einfache Art und Weise virtuelle Computer mit unterschiedlichen Betriebssystemen erstellt werden.



Im Anhang im Kapitel 19.5 gibt es in diesem Buch eine Anleitung, wie mit dem Hyper-V Manager ein virtueller Computer mit Ubuntu als Betriebssystem angelegt werden kann.

Falls Sie aber über die nötige Hardware verfügen und den nötigen Installationsaufwand betreiben wollen, dann steht es Ihnen natürlich auch offen, die Nodes auf mehrere physikalische Maschinen zu verteilen und zu orchestrieren. Die dazu nötigen Schritte und die Docker-Komman-

dos bleiben im Grunde genommen die gleichen wie bei der Arbeit mit den virtuellen Maschinen, die Ihnen in den folgenden Kapiteln vorgestellt werden.

Eine sehr bequeme Art, sich mit Docker und seinen Kommandos im Swarm Mode vertraut zu machen, ist ein Internet-Projekt mit dem Namen „Play with Docker". Es handelt sich hierbei um eine Webseite, die es erlaubt, in einer Trainingsumgebung simulierte Instanzen als Nodes zu erstellen und für eine begrenzte Zeit damit zu üben. Im Anhang im Kapitel 19.6 erhalten Sie nähere Informationen über den Zugang und die Arbeit mit dieser Web-Applikation.

### 15.5.1 Virtuelle Nodes mit Docker Machine

Damit Sie mithilfe von Docker Machine virtuelle Maschinen aufsetzen können, muss das Tool auf Ihrem Arbeitsrechner installiert sein.

In den meisten Fällen wird es zusammen mit Docker bzw. mit Docker Desktop automatisch installiert. Mittlerweile gibt es aber Versionen von Docker Desktop bzw. Docker Engine, wo das nicht der Fall ist.

Öffnen Sie ein Shell-Fenster und geben Sie das folgen Kommando ein:

```
1 docker-machine version
```

Wird als Antwort eine Versionsnummer angezeigt, ist alles in Ordnung. Erscheint aber eine Fehlermeldung wie `'command not found'` oder ähnlich (je nach Shell), dann müssen Sie Docker Machine manuell nachinstallieren.

Der Installationsprozess für die verschiedenen Betriebssysteme wird im Anhang im Kapitel 19.4 dieses Buches beschrieben.

Falls Sie auf einem System mit Windows 10 neben der PowerShell auch die Ubuntu Shell eingerichtet haben, dann muss dort Docker Machine bei Bedarf zusätzlich installiert werden, so wie es für die Installation unter Linux im Kapitel 19.4 beschrieben ist.

## 15.5.2 Docker Machine unter Windows

### *15.5.2.1 Vorbereitung von Hyper-V*

Docker Desktop setzt unter Windows 10 Hyper-V ein, um virtuelle Maschinen zu betreiben. Grundsätzlich könnte man unter Windows 10 auch Virtual Box installieren und damit virtuelle Systeme einrichten. Der gleichzeitige Betrieb beider Systeme zum Betreiben virtueller Maschinen ist aber problematisch und wird deshalb nicht empfohlen.

Das ist der Grund, weshalb wir beim Start von `docker-machine` unter Windows den Treiber von Hyper-V als Parameter angeben.

Beim Einsatz von Hyper-V ist es aber notwendig, dass mithilfe des Hyper-V Managers ein sogenannter virtueller Switch eingerichtet wird.

Darüber werden die virtuellen Computer auf einem Host mit virtuellen oder physikalischen Netzwerken verbunden.

Wir geben im Suchfeld der Windows-Taskleiste die Zeichenfolge „Hyper-V“ ein, um den Hyper-V Manager zu starten. Im Ergebnisfenster klicken wir dann auf das Hyper-V Icon (Abb. 15.17).

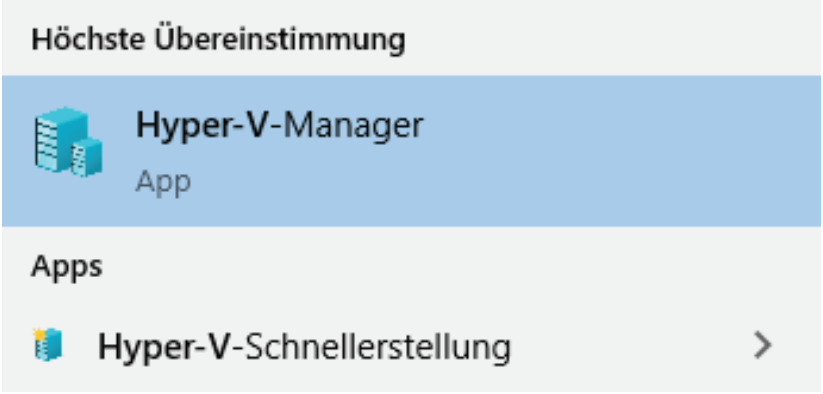



**Abb. 15.17** Hyper-V Manager ausführen

Das Hauptfenster der Hyper-V Manager-Applikation wird angezeigt (Abb. 15.18).

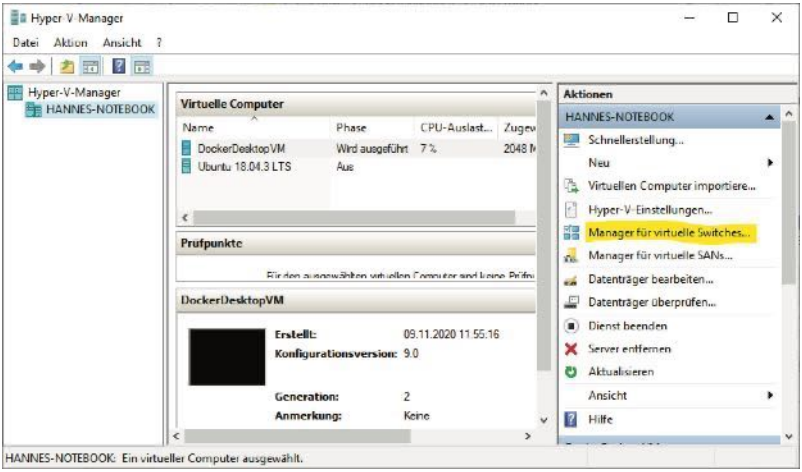



**Abb. 15.18** Hauptfenster des Hyper-V Managers

Im rechten Bereich des Programmfensters befindet sich die Gruppe *Aktionen*. Wählen Sie dort das Symbol MANAGER FÜR VIRTUELLE SWITCHES... aus.

Damit öffnet sich das Fenster „MANAGER FÜR VIRTUELLE SWITCHES“ (Abb. 15.19).

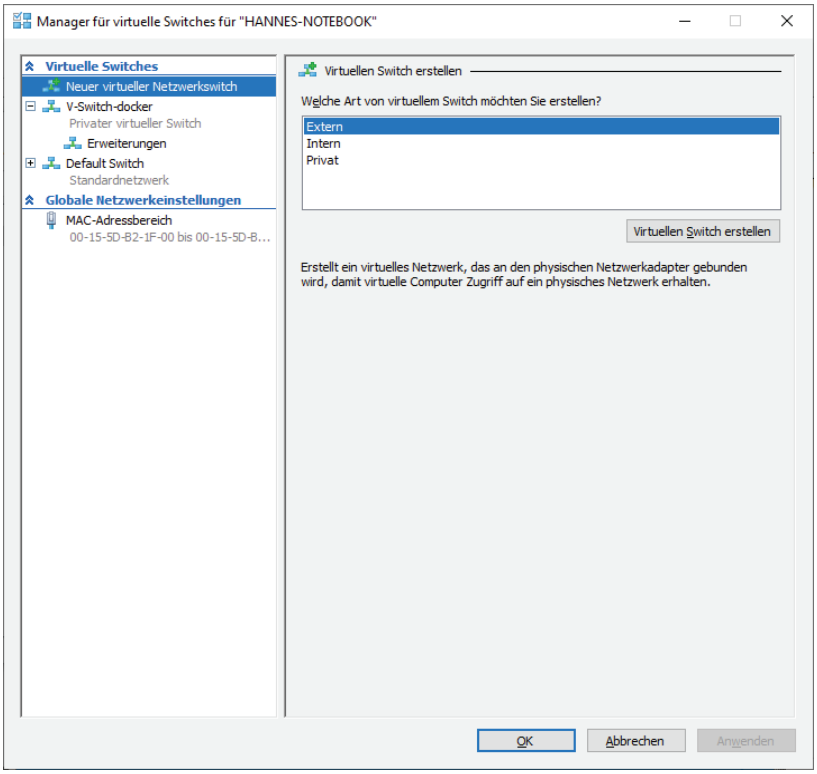



**Abb. 15.19** Hyper-V Managers: das Fenster „Manager für virtuelle Switches“



Markieren Sie dort in der linken Tabelle den Eintrag ‚Neuer virtueller Switch‘ und markieren Sie rechts *Extern* als Art für den neuen virtuellen Switch.

Mit einem Klick auf die Schaltfläche [Virtuellen Switch erstellen] legt Hyper-V den neuen virtuellen Switch an.

Im rechten Teil des Fensters können danach die Eigenschaften für den neuen virtuellen Switch eingegeben werden (Abb. 15.20).

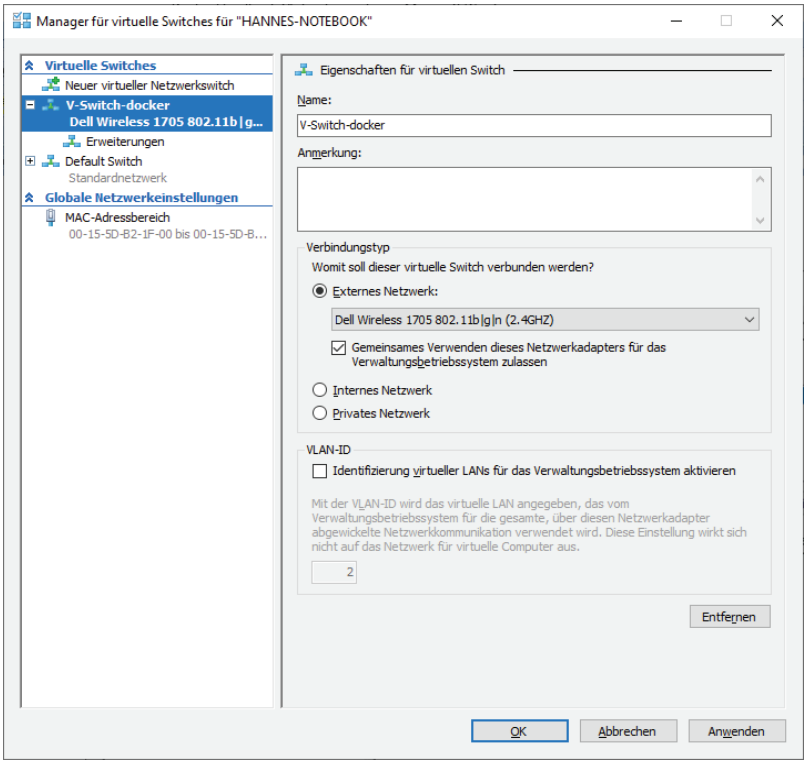



**Abb. 15.20** Hyper-V Manager: Eigenschaften für virtuelle Switches

Im Beispiel erhält der Switch den Namen „*V-Switch-docker*“. Er wird mit einem externen Netzwerk verbunden. Hier im Beispiel wurde ein Dell Wireless-Adapter ausgewählt. Die gemeinsame Verwendung des Netzwerkadapters für das Verwaltungs-Betriebssystem wird im Beispiel auch zugelassen.

Nachdem die Einstellungen mit [OK] übernommen worden sind, wird eine Warnung angezeigt, dass die Netzwerkkonnektivität unterbrochen werden kann. Wir bestätigen das mit [JA] (Abb. 15.21).

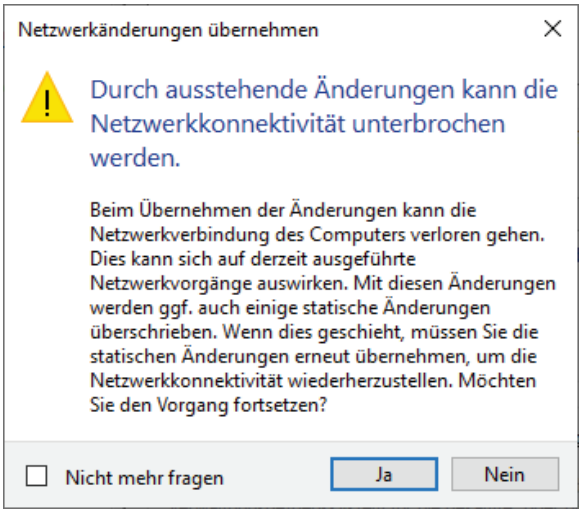



**Abb. 15.21** Hyper-V Warnung zu Netzwerkänderungen

Damit die durchgeführten Änderungen wirksam werden und der virtuelle Switch auch wirklich korrekt arbeitet, empfiehlt es sich, den Rechner neu zu starten.

## 15.5.3 Docker Swarm mit Manager und Worker Nodes

### *15.5.3.1 Manager Node auf virtueller Maschine erstellen*

Als erstes erstellen wir auf einer virtuellen Maschine einen Manager Node. Unter Windows ist es dabei nötig, dass wir Administratorrechte haben. Um eine PowerShell als Administrator zu starten, geben Sie im Suchfeld der Windows-Taskleiste die Zeichenfolge „PowerShell“ ein. In der Ergebnisliste, die daraufhin angezeigt wird, klicken Sie mit der rechten Maustaste auf den passenden Eintrag und wählen aus dem Kontextmenü den Menüpunkt Als Administrator ausführen (Abb. 15.22).

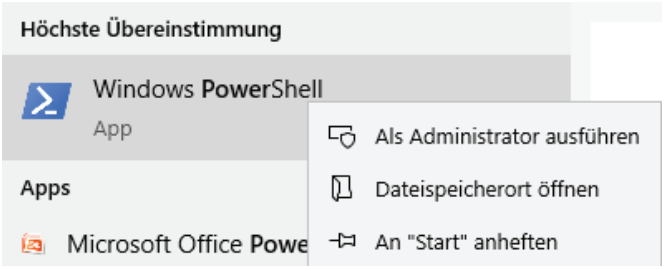



**Abb. 15.22** PowerShell als Administrator ausführen

Aus der Shell heraus erstellen wir jetzt eine virtuelle Maschine mithilfe des Kommandos `docker-machine create`.

Bei einer Installation von Docker Desktop unter Windows 10 mit Hyper-V sieht das Kommando folgendermaßen aus:

```
> docker-machine create -d hyperv '
--hyperv-virtual-switch V-Switch-docker manager1
```

Mit dem Parameter `-d` beziehungsweise `--driver` wird angegeben, mit welchem Treiber die virtuelle Maschine arbeiten soll. Das obige Kommando gibt `hyperv` an. Darauf folgt mit dem Parameter `--hyperv-virtual-switch` die Angabe des Namens für den virtuellen Switch. Wir haben gerade mit dem Hyper-V Manager einen virtuellen Switch mit dem `Namen V-Switch-docker` erstellt und der wird bei diesem Beispiel verwendet. Als letzten Parameter übergeben wir den von uns gewünschten Namen für den Node, im Beispiel soll er `manager1` lauten.

Wenn das Kommando ordnungsgemäß funktioniert hat, und Sie anschließend den Hyper-V Manager starten, dann wird die neue virtuelle Maschine mit dem Namen manager1 in der Liste der aktuell laufenden virtuellen Maschinen mit angezeigt.

Wenn Sie den neuen Eintrag in dieser Liste markieren, dann erhalten Sie im unteren Bereich des Hyper-V Managers zusätzliche Informationen für diese Auswahl. Im Screenshot wurde das Register-Netzwerk für

den virtuellen Computer `manager1` aktiviert. Hier sehen Sie, dass der virtuelle Switch V-Switch-docker verwendet wird und dass der virtuelle Computer in diesem Fall unter der IP-Adresse 192.168.57.39 erreicht werden kann (Abb. 15.23).

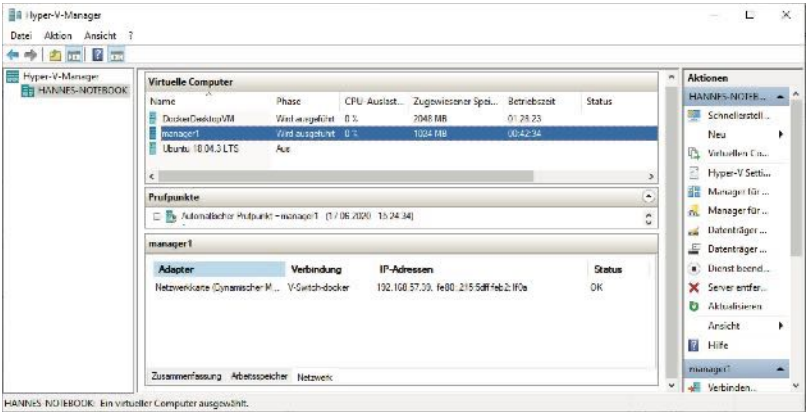



**Abb. 15.23** Eine virtuelle Maschine im Hyper-V Manager

Wenn als Treiber für die virtuelle Maschine VirtualBox verwendet werden soll, z.B. auf Linux oder Mac-OS Systemen, oder es wurde Docker Toolbox auf einem älteren Windows-System installiert, dann sieht das Kommando wie folgt aus:

```
1  > docker-machine create -d virtualbox manager1
```

VirtualBox muss dazu auf dem System installiert sein. Unter Windows 10 kann Hyper-V aber nicht zusammen mit VirtualBox aktiv sein. Die beiden Systeme sind nicht kompatibel.

Sehen wir uns jetzt an, welche virtuellen Maschinen auf unserem System aktiv sind. Hier das Shell-Kommando, um die aktuelle Liste der virtuellen Maschinen auszugeben:

```
1  > docker-machine ls
```

Um auf eine Kommando-Shell für die neue virtuelle Maschine zugreifen zu können, müssen wir uns mit der virtuellen Maschine verbinden. Hier das Kommando, um sich an der virtuellen Maschine mit dem Namen `manager1` anzumelden und dort eine SSH Shell zu starten:

```
1  > docker-machine ssh manager1
```

Der nächste Screenshot zeigt das Ergebnis der hier beschriebenen Kommandos (Abb. 15.24).

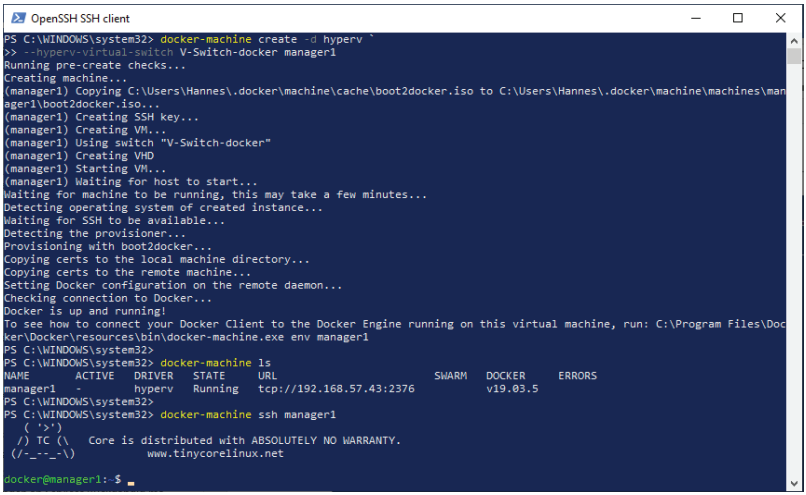



**Abb. 15.24** Docker-Machine: eine virtuelle Maschine erzeugen

Damit sind wir soweit, dass wir einen neuen Swarm auf dem Node `manager1` erstellen können. Das erledigen wir mit dem folgenden Kommando in der SSH Shell auf der dazugehörigen virtuellen Maschine:

```
1  docker swarm init --advertise-addr <MANAGER_IP>
```

Damit wird der Swarm Mode von Docker auf dieser Maschine initialisiert (Abb. 15.25). Die Advertise-Adresse des Manager Nodes kann, wie weiter oben bereits gezeigt, mithilfe des Hyper-V Managers ermittelt

werden oder mithilfe des Kommandos `docker-machine ls` (siehe Abb. 15.24).

```
1  $ docker swarm init --advertise-addr 192.168.57.43
```

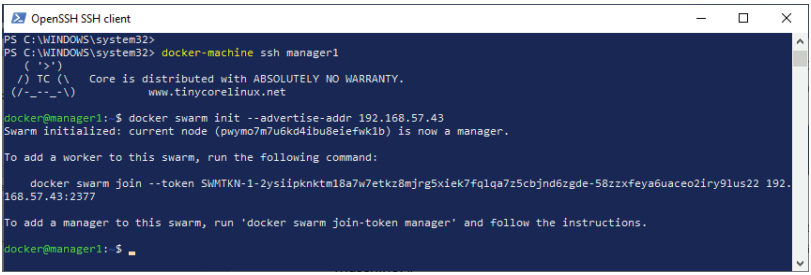



**Abb. 15.25** Den Swarm Mode auf einer virtuellen Maschine initialisieren

Wie Sie ja bereits wissen, gibt Docker mit dem Init-Kommando in der Ausgabe das Kommando bekannt, mit dem ein Worker Node dem Swarm zugefügt werden kann.

Falls man diese Information zu einem späteren Zeitpunkt wieder braucht, kann man den Join Token, mit dem man einen weiteren Node dem Swarm hinzufügen kann, auch nachträglich wieder herausfinden. Docker bietet dafür das folgende Kommando:

```
1  docker swarm join-token worker | manager
```

Da es sich bei diesem Kommando um ein Cluster Management-Kommando handelt, muss es logischerweise innerhalb eines Manager Nodes aufgerufen werden.

### *15.5.3.2 Worker Node erstellen*

Damit wir einen echten Swarm erstellen können, erzeugen wir jetzt eine neue virtuelle Maschine. Dazu muss wieder das Kommando docker-machine create mit den passenden Parametern gestartet werden.

Unter Windows muss das wieder in einer PowerShell, die auch hier mit Administrator-Rechten gestartet worden ist, eingegeben werden. Um die Arbeit mit den folgenden Beispielen etwas komfortabler zu gestalten, starten wir eine weitere Kommando-Shell. Das hat den Vorteil, dass wir dann die virtuellen Maschinen jeweils innerhalb einer eigenen Shell laufen lassen können. Das ermöglicht uns den einfachen Wechsel zwischen den virtuellen Maschinen.

Das Shell-Kommando, um unter Windows die virtuelle Maschine mit dem Hyper-V-Treiber und mit dem Namen `worker1` zu erstellen, sieht jetzt so aus:

```
1 > docker-machine create -d hyperv '
2 --hyperv-virtual-switch V-Switch-docker worker1
```

So sieht die Variante mit dem VirtualBox-Treiber aus:

```
1 > docker-machine create -d virtualbox worker1
```

Nachdem die virtuelle Maschine erstellt worden ist, verbinden wir uns wieder mit ihr:

```
1 > docker-machine ssh worker1
```

Wir befinden uns daraufhin in der Kommando-Shell der virtuellen Maschine mit dem Namen worker1 und können von hier aus einem Swarm beitreten.

Wie bereits weiter oben erwähnt wurde, können wir bei Bedarf das dafür benötigte Join-Kommando in der Shell des *Manager Nodes* abfragen.

Soll eine Maschine als *Worker* Node beitreten, so fragen wir das dafür nötige Kommando wie folgt ab (wie gesagt, in der Shell des Manager Nodes `manager1`):

```
1 $ docker swarm join-token worker
```

Soll ein neuer Node dagegen in der Rolle eines *Manager* Nodes eingebunden werden, dann sieht das Kommando so aus:

```
1  $ docker swarm join-token manager
```

Der folgende Screenshot zeigt den Aufruf von beiden Varianten (Abb. 15.26):

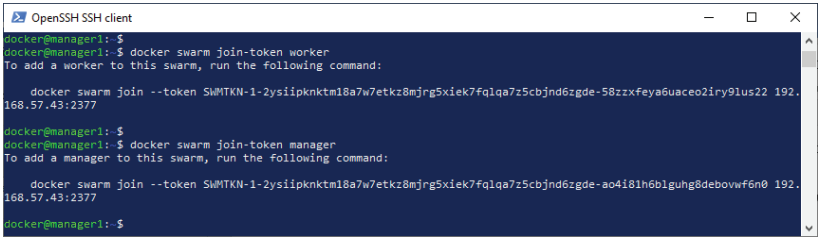



**Abb. 15.26** Abfrage des Join Token auf einem Manager Node

Das Kommando kann jetzt problemlos aus der Shell des Manager Nodes in die Zwischenablage kopiert werden (zum Beispiel mit der Tastenkombination <STRG + C>). Danach wechseln wir zur Shell des Nodes, welcher dem Swarm zugefügt werden soll, fügen es dort ein (zum Beispiel mit der Tastenkombination <STRG + V>) und führen es aus (Abb. 15.27).

Das Kommando sieht in unserem Beispiel für den Worker Token folgendermaßen aus:

```
1  $ docker swarm join --token SWMTKN-1-2ysiipknktm18a7w7etkz8mjrg5
2  xiek7fqlqa7z5cbjnd6zgde-58zzxfeya6uaceo2iry9lus22
3  192.168.57.43:2377
```

Der Token und auch die IP werden im Normalfall natürlich jedes Mal andere Werte annehmen.


**Abb. 15.27** Einen Node einem Swarm zufügen

### 15.5.3.3. *Das Cluster untersuchen*

Untersuchen wir jetzt erst einmal unseren neuen Swarm.

Dazu begeben wir uns wieder in die Shell unseres Manager Nodes `manager1`.

Dort lassen wir uns von Docker die aktuellen Informationen mit dem Kommando `docker info` anzeigen (Abb. 15.28).

```
1  $ docker info
```

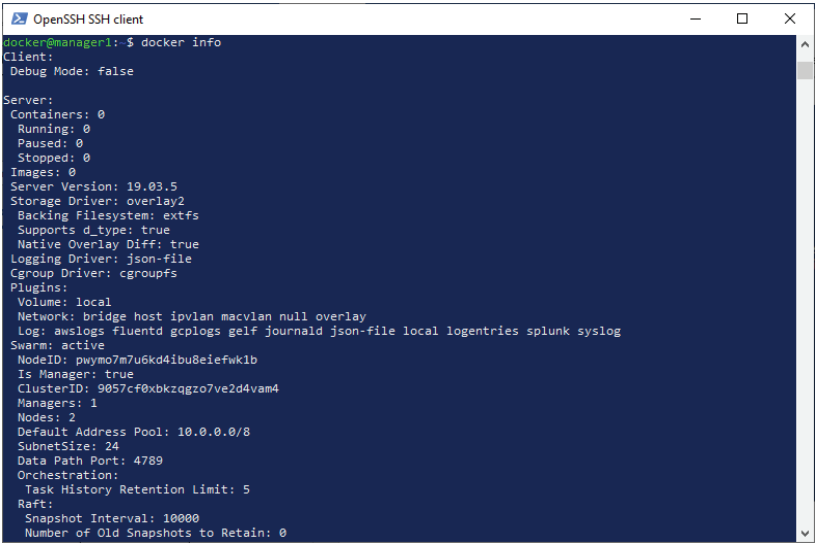



**Abb. 15.28** Docker-Info im Manager Node eines Swarms

Im obigen Screenshot können Sie unter anderem ablesen, dass der Swarm Mode aktiv ist und dass der Swarm aus 2 Nodes besteht, von denen einer die Manager-Rolle innehat. Man sieht auch, dass der aktuelle Node der Manager ist. Es wird auch die Node ID und die Cluster ID wiedergegeben.

Wir lassen uns auch noch eine Liste der Nodes in diesem Cluster ausgeben (Abb. 15.29):

```
1  $ docker node ls
```

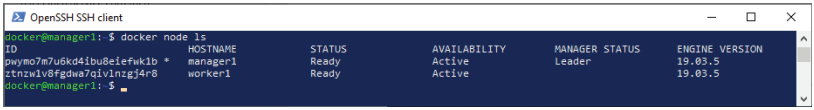



**Abb. 15.29** Node ls-Kommando bei einem Swarm mit zwei Nodes

Das `node ls`-Kommando funktioniert ebenfalls nur auf einem Manager Node.

Die erste Spalte zeigt die ID der Nodes an. Der node mit einem * neben der Node ID ist der gerade aktive Node, d.h. mit dem bin ich verbunden.

Es folgen die Spalten mit den Informationen zum Host-Namen, dem Status und der Verfügbarkeit der Nodes.

In der Spalte MANAGER STATUS kennzeichnet der Eintrag *Leader* die Zeile für den leitenden Manager.

### *15.5.3.4 Übungsaufgabe: den Swarm erweitern*

Bis jetzt war doch alles recht einfach, oder? Die nächste Übungsaufgabe sollte deshalb kein Problem darstellen:

Erweitern Sie unseren Swarm um einen weiteren Worker Node mit dem Namen `worker2`.

Untersuchen Sie das erweiterte Cluster.

***Lösung:***

1. Neue PowerShell als Administrator starten oder die CSS-Shell von einem der beiden laufenden Nodes verlassen. Das Linux-Kommando dafür lautet

```
$ exit
```

2. Eine weitere virtuelle Maschine mit dem Namen `worker2` erstellen:

```
> docker-machine create -d hyperv '
--hyperv-virtual-switch V-Switch-docker worker2
```

3. Mit der virtuellen Maschine verbinden:

```
> docker-machine ssh worker2
```

4. Die neue virtuelle Maschine dem Swarm als Worker Node hinzufügen. Das folgende Kommando ist nur ein Beispiel. Sie müssen das Kommando so verwenden, wie es Ihnen auf dem Manager Node mit dem Kommando `docker swarm join-token worker` vorgegeben wird:

```
$ docker swarm join --token SWMTKN-1-2ysiipknktm18a7w7etkz8mjrg5
xiek7fqlqa7z5cbjnd6zgde-58zzxfeya6uaceo2iry9lus22
192.168.57.43:2377
```

5. Auf die Shell des Nodes `manager1` wechseln. Dort die folgenden Kommandos eingeben, um den Swarm zu untersuchen:

```
$ docker info
$ docker node ls
```

### *15.5.3.5 Dem Swarm Services hinzufügen*

Bis jetzt haben wir nur die Laufzeitumgebung für eine verteilte Applikation geschaffen. Jetzt fehlen noch die Container mit den eigentlichen Anwendungen. Die aktivieren wir wieder, wie wir das schon beim Single Node Swarm erledigt haben, mit dem Docker-Kommando `docker service create`.

Als Beispiel wollen wir wieder einen Redis Service mit drei Tasks erstellen. Das Kommando dazu muss auf dem Manager Node des Swarms ausgeführt werden (Abb. 15.30):

```
$ docker service create --name my_redis --replicas=3 redis
```


**Abb. 15.30** Ein Redis Service mit drei Tasks

Wenn wir uns den Prozess-Status der Tasks für diesen Service anzeigen lassen, dann erhalten wir dabei unter anderem die Information, auf welchem Node die Tasks jeweils ausgeführt werden (Abb. 15.31).

```
$ docker service ps my_redis
```

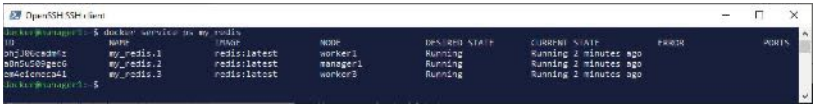



**Abb. 15.31** Prozess-Status des Redis Services mit drei Tasks

Wir können jetzt sehr schön sehen, wie die drei Task auf die drei Nodes des Swarms aufgeteilt worden sind.

Erweitern wir unser Cluster um einen NGINX Service mit drei Tasks

```
$ docker service create --name my_nginx --replicas=3 nginx
```

und lassen uns den Status aller Tasks für die beiden Services anzeigen (Abb. 15.32):

```
$ docker service ps my_redis my_nginx
```

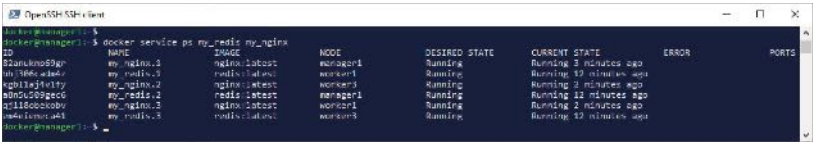



**Abb. 15.32** Prozess-Status des Redis und des NGINX Services mit jeweils drei Tasks

Auch die drei Tasks des NGINX Service hat Docker auf die drei Nodes des Swarms aufgeteilt.

### 15.5.4 Docker-Kommandos für Multi Node Swarms

Bei allen Kommandos, die hier folgen, handelt es sich um Cluster Management-Kommandos. Das bedeutet, diese Kommandos können nur aus einem Manager Node heraus ausgeführt werden.

Ausgabe der Tasks in einem Node.

Syntax:

```
1  > docker node ps [OPTIONS] [NODE...]
```

Standard ohne Angabe eines Nodes werden die Tasks des aktuellen Nodes aufgelistet.

Als Option ist auch hier unter anderem die Angabe `--format` möglich, um eine formatierte Ausgabe im Go-Format zu erhalten.

Der Screenshot zeigt die Anwendung des Kommandos einmal ohne Angabe eines Nodes, dann mit Angabe des Node-Namens und zuletzt mit der Angabe der ersten Zeichen der Node ID. Das erste Kommando, docker node ls, wurde eingegeben, um die Node ID herauszufinden (Abb. 15.33).

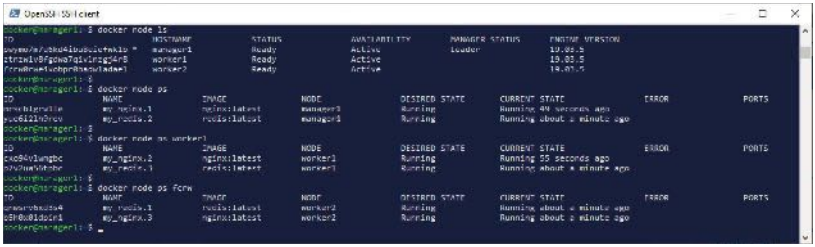



**Abb. 15.33** Anzeige des Task-Status von verschiedenen Nodes

Einen Swarm verlassen:

Soll ein Node den Swarm verlassen, so gibt man in der Shell dieses Nodes das folgende Kommando ein:

```
1  > docker swarm [--force] leave
```

Wenn ein Node den Swarm verlässt, dann beendet die Docker Engine für diesen Node den Swarm Mode. Es werden diesem Node keine Tasks mehr zugewiesen.

Wird das leave-Kommando bei einem Manager Node aufgerufen, dann wird es nicht ausgeführt und eine Meldung wird ausgegeben. Das kann durch die Angabe der Option `--force` unterdrückt werden. Verlässt der letzte Manager Node einen Swarm, dann kann auf diesen Swarm nicht weiter zugegriffen werden. Um unerwünschtes Verhalten zu verhindern, sollte der Manager Node als letztes den Swarm verlassen, so wie das ein guter Kapitän bei seinem Schiff tun würde.

Wenn ein Node den Swarm verlassen hat, dann kann auf einem Manager Node des Swarms das Kommando `docker node rm` für diesen Node aufgerufen werden, um ihn von der Node-Liste des Swarms zu entfernen.

Ein Node kann aus der Node-Liste des Manager Nodes erst entfernt werden, wenn dieser den Status ‚Down' hat, also wenn dieser Node den Swarm verlassen hat.

Entfernen eines Nodes aus einem Swarm:

Das Kommando, das einen Node aus einem Swarm entfernt, hat folgende Syntax:

```
1 > docker node rm [OPTIONS] NODE [NODE...]
```

Wenn Sie das Kommando ohne weitere Option für einen aktiven Node aufrufen, erhalten Sie allerdings eine Fehlermeldung (Abb. 15.34):


**Abb. 15.34** Fehlermeldung beim Versuch einen aktiven Node zu entfernen

Der Node sollte normalerweise vorher ordentlich heruntergefahren werden. Es ist aber möglich, Docker zu zwingen einen Node zu entfernen. Dafür gibt es bei diesem Kommando die Option `-f` oder auch `--force`.

Wenn das Kommando, wie im nächsten Beispiel eingegeben wird, dann wird es auch ohne Fehlermeldung ausgeführt:

```
1 > docker node rm --force worker2
```

Beförderung eines Nodes zum Manager:

Falls ein Node in der Rolle eines Worker Nodes zu einem Cluster hinzugefügt wurde, kann dieser bei Bedarf jederzeit zu einem Manager Node befördert, d.h. heraufgestuft werden.

```
1 > docker node promote <NODE>
```

Hier ein Beispiel, um den Node `'worker1'` zu einem Manager zu „befördern". Anschließend zeigen wir die Liste der Nodes an (Abb. 15.35):

```
1 > docker node promote worker1
2 > docker node ls
```

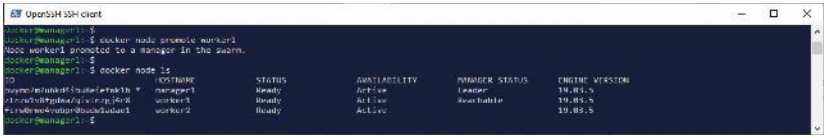



**Abb. 15.35** Heraufstufen eines Worker Nodes zum Manager

Node `'worker1'` hat hier den Manager-Status *Reachable* angenommen. Das bedeutet: Falls der Node ausfällt, der aktuell den Status *Leader* hat, dann steht dieser Node für die Wahl des neuen Managers zur Verfügung.

Herabstufung eines Manager Nodes:

Ein Manager Node kann mit dem nächsten Kommando auch wieder „degradiert", d.h. herabgestuft, werden.

```
1  > docker node demote <NODE>
```

Auch das soll noch mit einem praktischen Beispiel demonstriert werden. Wir degradieren `'worker1'` wieder und sehen uns das Ergebnis in der Liste der Nodes an (Abb. 15.36):

```
1  > docker node demote worker1
2  > docker node ls
```

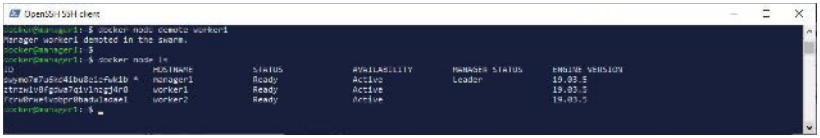



**Abb. 15.36** Herabstufen eines Manager Nodes zum Worker

## 15.6 Docker Configs - verteilte Konfigurationen

Docker Configs bieten die Möglichkeit, unkritische Informationen, wie zum Beispiel HTML-Seiten, zu speichern, ohne dass Konfigurationsdateien in den Images für Container eingebunden werden müssen oder

Informationen über Umgebungsvariablen zur Verfügung gestellt werden. Docker Configs sind aber nicht verschlüsselt und werden direkt in das Dateisystem eines Containers montiert.

Configs stehen allerdings nicht bei einzelnen Containern zu Verfügung, sondern sie sind nur für Swarm Services gedacht.

Einem Service können solche Docker Configs jederzeit hinzugefügt und auch wieder weggenommen werden. Configs können als Werte sowohl Zeichenketten als auch Binäre Daten enthalten.

Wenn eine Konfiguration einem Swarm zugefügt wird, dann übernimmt diese der Swarm Manager und speichert die enthaltenen Daten innerhalb der Raft-Datenbank. Die ist wiederum verschlüsselt, und wird in allen Manager Nodes des Swarm repliziert. Damit stehen die Informationen von Docker Configs allen Manager Nodes gleichermaßen zur Verfügung.

Bevor wir uns mit den verschiedenen Kommandos zum Umgang mit Docker Configs befassen, räumen wir unseren Swarm erst einmal auf und löschen dazu die aktiven Services, falls noch welche laufen. Um zum Beispiel die im letzten Kapitel erstellten Services `'my_nginx'` und `'my_redis'` zu löschen, benutzen wir dieses Kommando:

```
1 $ docker service rm my_nginx my_redis
```

Jetzt können wir die Arbeit mit Docker Configs beginnen.

### 15.6.1 Docker-Konfiguration erstellen

Im ersten Schritt erstellen wir eine Docker-Konfiguration. Dazu legen wir eine Datei an, welche die Konfigurationsinformationen enthält. Wir wechseln wieder in die Shell des Manager Nodes `'manager1'`. Dort erstellen wir eine Datei mit beliebigem Inhalt. Das kann zum Beispiel mit dem `echo`-Kommando von Linux durchgeführt werden. Das Kommando echo gibt normalerweise den Text, der als Parameter angegeben wird, am Bildschirm aus.

Durch die Verwendung des Umleitungszeichens > kann die Ausgabe des Textes in eine Datei mit dem angegebenen Dateinamenamen bewirkt werden, anstatt dass es auf dem Monitor oder im Fenster einer Shell ausgegeben wird.

```
1 $ echo "Here is my config information" > test.cfg
```

Das so eingegebene Kommando erstellt eine Datei mit dem Namen `'test.cfg'` und schreibt dort die Zeichenkette „Here is my config information“ hinein.

Mit dem `cat`-Kommando können wir uns den Inhalt dieser Datei wieder ausgeben lassen.

```
1 $ cat test.cfg
```

Diese Datei soll jetzt unsere Konfigurationsdatei sein. Für den Service einer echten Anwendung wird man natürlich eine Datei mit sinnvollerem Inhalt verwenden (z.B. eine HTML-Datei). Aber für unsere Übungszwecke reicht diese Datei vollkommen aus.

Um aus der Datei eine Docker-Konfiguration zu erzeugen, stellt Docker das Kommando `docker config create` bereit. Hier zunächst die Syntax:

```
1 $ docker config create [<OPTIONS>] <CONFIG> <FILE>|-
```



Mit `<CONFIG>` geben wir der Konfiguration einen Namen. Mit dem letzten Parameter übergeben wir einen Dateinamen oder wir setzen ein Minus-Zeichen (–) ein. Bei Angabe des Minus-Zeichens wird STDIN (die Tastatur) als Eingabe für die Zeichenkette mit der Konfigurationsinformation verwendet.

Das folgende Beispiel zeigt ein Kommando, bei dem die Konfiguration aus der Datei `'test.cfg'` erzeugt wird:

```
1 $ docker config create my_config test.cfg
```

Und hier die Variante mit STDIN als Eingabe:

```
$ echo "Here is my config information" | docker config create my_
config -
```

Wir erzeugen unseren Konfigurationsstring mit einem echo-Kommando. Durch das Pipe-Zeichen (|) geben wir an, dass die Ausgabe von `echo` als STDIN-Eingabe für das darauffolgende Kommando hergenommen werden soll. Dann folgt das eigentliche `docker config create`-Kommando das als letzten Parameter das Minus-Zeichen erhält, um damit zu bewirken, dass STDIN als Eingabe verwendet wird.

Das Ergebnis ist in diesem Fall bei beiden Varianten des Kommandos das gleiche.

Wie bei den meisten Docker-Objekten gibt es auch für Docker Configs ein `ls`-Kommando, um eine Liste der aktuellen Docker-Konfigurationen anzuzeigen:

```
$ docker config ls
```

Der Screenshot zeigt obige Aktionen, die in der Shell des Manager Nodes ausgeführt worden sind (Abb. 15.37):

- Konfigurationsdatei erstellen.
- Konfigurationsdatei anzeigen.
- Docker Config erstellen.
- Liste der Docker Configs anzeigen.

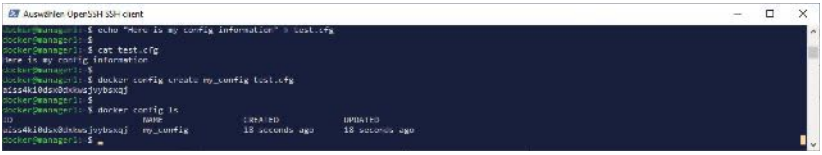



**Abb. 15.37** Docker Config erstellen

Um Konfigurationen zu untersuchen, gibt es das `docker config inspect`-Kommando:

```
1 $ docker config inspect [<OPTIONS>] <CONFIG> [<CONFIG>]
```

Hier ein einfacher Aufruf des Inspect-Kommandos (Abb. 15.38):

```
1 $ docker config inspect my_config
```


**Abb. 15.38** Docker Config untersuchen

Docker verbirgt hier bei den Informationen die eigentlichen Daten, um die Ausgabe nicht unnötig lang werden zu lassen. Von unserer Konfiguration sehen wir bei der Ausgabe durch diese Form aber recht wenig. Besser wird es, wenn wir das Kommando durch die Option `--pretty` ergänzen. Jetzt ist der Inhalt der Konfiguration `my_config` auch sichtbar (Abb. 15.39).

```
1 $ docker config inspect --pretty my_config
```


**Abb. 15.39** Docker Config inspect Kommando mit Option --pretty

Um herauszufinden, ob die Konfiguration auch auf einem anderen Node verfügbar ist, müssen wir einen unserer drei Swarm Nodes noch einmal zum Manager befördern. Als Beispiel soll hier der Node der virtuellen Maschine `worker1` heraufgestuft werden. Dazu geben wir im Node `manager1` das benötigte Kommando ein:

```
1 $ docker node promote worker1
```

Nun wechseln wir in die Kommando-Shell von `worker1`. Von dort aus versuchen wir auf die Konfigurationsdaten mit den Kommandos `docker config ls` und `docker config inspect` zuzugreifen (Abb. 15.40).

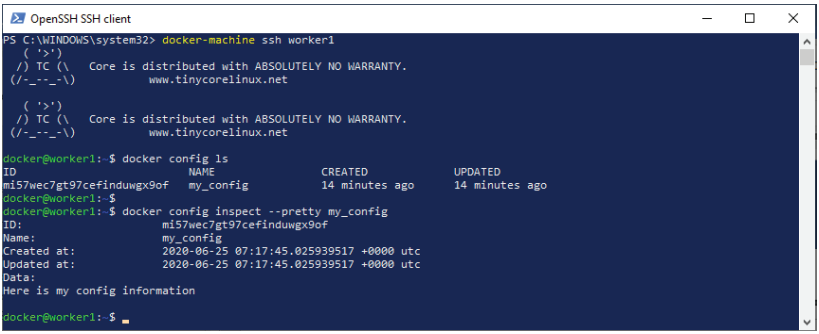



**Abb. 15.40** Zugriff auf eine Docker Config aus einem anderen Node

Es kann also problemlos aus einem anderen Node des Swarms, der die Manager-Rolle besitzt, auf die Konfigurationen des Swarm zugegriffen werden. Man kann auch aus jeder Manager Shell heraus neue Konfigurationen für den Swarm erstellen.

### 15.6.2 Docker Configs einem Service übergeben

Die Konfiguration ist jetzt für den Swarm verfügbar, wird aber bis jetzt noch von niemandem benutzt.

Konfigurationen haben ja den Zweck, Informationen an Docker Container weiterzugeben. Im Swarm Modus werden Container als Elemente von Services instanziiert. Beim Erstellen eines Service können eine oder mehrere Konfigurationen als Parameter übergeben werden.

Der Parameter hat den Namen `–config`, auf diesen folgt der Name der Konfiguration.

Das folgende Beispiel erzeugt einen Docker Service mit dem Namen `'my_redis'`. Dieser instanziieret drei Container aus dem Image `'redis:alpine'` vom Docker Hub und gewährt diesen den Zugriff auf die Konfiguration mit dem Namen `'my_config'`.

```
$ docker service create --name my_redis --config my_config \
--replicas=3 redis:alpine
```

Damit wir sehen, ob und wo die Konfigurationen in den Containern angelegt wurden, versuchen wir einen Blick in diese zu werfen.

Um die Beispielkommandos allgemeingültiger zu gestalten, fragen wir die ID des Containers mit dem Namen `'my_redis'` mit einem Docker-Kommando ab und setzen diesen Ausdruck im Kommando anstelle der Container ID ein.

```
$ docker ps --filter name=my_redis -q
```

Diese Konstruktion übergeben wir jetzt anstelle der Container ID an das Docker Container-Kommando, mit dem wir in einem Container mit dieser ID einen Befehl ausführen können.

Mit dem ersten Beispiel führen wir das Linux-Kommando `ls` aus und lassen uns aus dem `root`-Verzeichnis die Dateiinformationen der Datei `'/my_config'` ausgeben.

```
$ docker container exec $(docker ps --filter name=my_redis -q)\
 ls -l /my_config
```

Man kann gut sehen, dass als Mount Point der Konfigurationsname auf dem `root`-Verzeichnis des Containers angelegt wurde. Diese Position ist bei Linux Containern Standard.

Bei Windows Containern werden Configs in das Verzeichnis `'C:\ProgrammData\Docker\configs'` montiert. Es werden auf diese Dateien symbolische Links erzeugt, die sich standardmäßig im Root-Verzeichnis von Laufwerk C: befinden.

Das zweite Beispiel zeigt uns jetzt den Inhalt der Datei `'my_config'` an.

```
$ docker container exec $(docker ps --filter name=my_redis -q)\
 cat /my_config
```

Hier folgt noch ein Screenshot von der Ausführung der gerade vorgestellten Kommandos (Abb. 15.41).

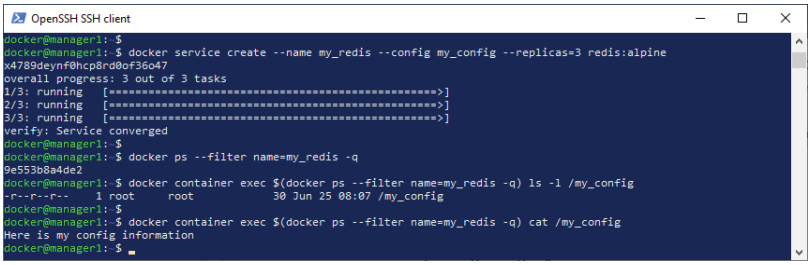



**Abb. 15.41** Docker-Konfigurationen an einen Service und dessen Container übergeben

Wieder wollen wir überprüfen, ob die Konfiguration auch in den Containern des Service, die auf den anderen Nodes laufen, verfügbar ist.

Zur Kontrolle wechseln wir diesmal in die Shell des Nodes `'worker2'`. Mit dem Kommando `docker ps` zeigen wir die laufenden Container an. Der Container des Service `'my_redis'` hat hier die ID a85e3b20ca7.

Noch einmal lassen wir uns den Inhalt der Konfiguration `'my_config'`anzeigen und, wie wir sehen können, ist der Wert hier ebenfalls vorhanden und sein Inhalt identisch zu dem in den anderen Containern auf den anderen Nodes, die hier auf virtuellen Maschinen laufen, die aber genauso gut physikalische Computer sein könnten (Abb. 15.42).

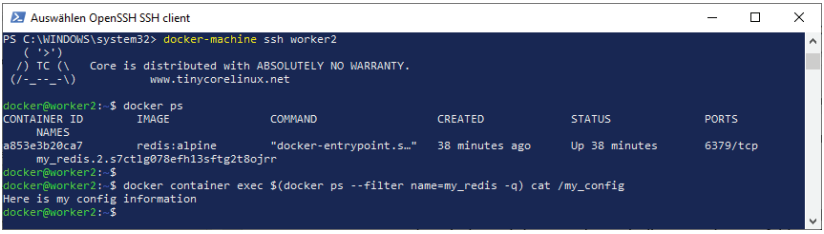



**Abb. 15.42** Docker-Konfigurationen auf verschiedenen Nodes eines Swarms

Wie immer räumen wir den Swarm zuletzt auf, bevor wir zum nächsten Kapitel weiter gehen.

Dazu verwenden wir das `docker config rm`-Kommando, mit dem man Docker-Konfigurationen wieder löscht:

```
1 $ docker config rm <CONFIG> [<CONFIG>]
```

Auch das ist ein Orchestrierungskommando und kann nur auf einem Manager Node ausgeführt werden. Wir verbinden uns deshalb wieder mit der Shell auf unserem Node `'manager1'`.

Für die Konfiguration `'my_config'` sieht das Lösch-Kommando so aus:

```
1 $ docker config rm my_config
```

Aber so führt es erst einmal zu einer Fehlermeldung. Docker sagt uns, dass die Konfiguration `'my_config'` noch vom Service `'my_redis'` benutzt wird und darum nicht gelöscht werden darf.

Vorher müssen für alle Services die Zugriffe auf die zu löschende Konfiguration entzogen werden. Bei einem aktiven Service verwendet man dafür das Kommando `docker service update` mit der Option `--config-rm`, gefolgt vom Namen der Konfiguration.

Beispiel:

```
1  $ docker service update --config-rm my_config my_redis
```

Jetzt erst lässt sich die Konfiguration ohne Problem entfernen (Abb. 15.43).

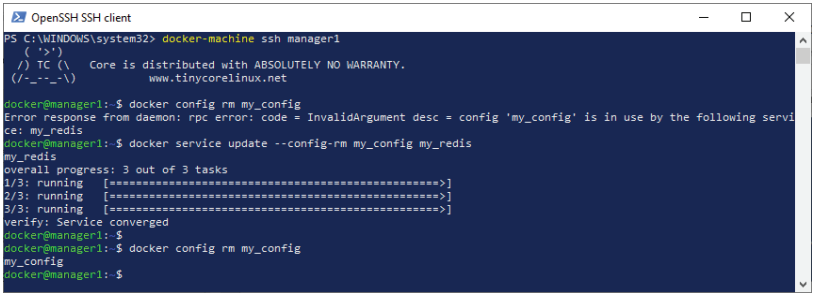



**Abb. 15.43** Eine Docker-Konfiguration wieder löschen

Den Service `'my_redis'` behalten wir noch für das nächste Kapitel und auch den Swarm mit seinen Nodes aus virtuellen Maschinen brauchen wir noch ein paar Mal und lassen sie deshalb weiter bestehen.

## 15.7 Secrets: sensitive Daten verstecken

Wie geht man bei der Arbeit unter Docker mit kritischen Daten um, die nicht für jedermanns Auge bestimmt sind, wie zum Beispiel Benutzernamen oder Passwörter?

Für diese Art Daten stellen die Docker Swarm Services sogenannte *Secrets* zur Verfügung. Die Daten von Secrets werden dabei durch sogenannte Blobs (Binary Large Objects) realisiert. Damit werden Informationen gehandhabt, die nicht unverschlüsselt über ein Netzwerk übertragen werden sollen oder die aus Dockerfiles oder aus dem Quellcode von Anwendungen ausgelesen werden könnten.

Docker Secrets ähneln im Grunde genommen ein wenig den Docker Configs, nur dass sie eben verschlüsselt sind. Secrets werden, so wie auch Configs, in der Raft-Datenbank eines Clusters gespeichert und über diese verteilt. Auf jedes Secret können ausschließlich die Docker Services zugreifen, denen dieser Zugriff ausdrücklich gewährt worden ist und nur solange diese Services auch ausgeführt werden.

Eine weitere Gemeinsamkeit mit Configs ist, dass Secrets nur den Swarm Services zu Verfügung stehen und bei einzelnstehenden Containern nicht eingesetzt werden können.

### 15.7.1 Docker Secrets erstellen

Die Aktionen, mit denen wir unser erstes Geheimnis als Secret erstellen, werden Ihnen wahrscheinlich sehr bekannt vorkommen.

Wieder erstellen wir eine Datei, welche die „geheime" Information enthält. Auch das passiert wieder in der Shell unseres Manager Nodes `'manager1'`. Dort erstellen wir noch einmal eine Datei mit beliebigem Inhalt.

```
1 $ echo "A very secret information" > mysecret.txt
```

Damit haben wir eine Datei `'mysecret.txt'` mit dem Inhalt „A very secret information" angelegt.



Zu Sicherheit sehen wir uns den Inhalt der Datei noch einmal an.

```
1 $ cat mysecret.txt
```

Um aus dieser Datei ein Docker Secret zu erstellen, nutzen wir jetzt das Docker-Kommando `docker secret create`. Zuerst sehen wir uns die Syntax an:

```
1 $ docker secret create [<OPTIONS>] <SECRET> <FILE>|->
```

Den Namen des Secrets übergeben wir mit der Option `<SECRET>`. Auch hier, wie beim Anlegen einer Konfiguration, übergeben wir als letztes Argument einen Dateinamen oder das Minus-Zeichen (–). Mit dem Minus-Zeichen wird wieder STDIN (die Tastatur) für die Eingabe verwendet.

Dieses Beispiel zeigt das Kommando, um aus der Datei `'mysecret.txt'` ein Secret zu erzeugen:

```
1 $ docker secret create my_secret mysecret.txt
```

Und hier die Variante mit STDIN als Eingabe:

```
1 $ echo " A very secret information" | \
2 docker secret create my_secret -
```

Wieder ist das Ergebnis bei beiden Varianten gleich.

Mit dem `ls`-Kommando sehen wir uns das neu erstellte Secret mit seinen Informationen in einer Liste an:

```
1 $ docker secret ls
```

Der Screenshot zeigt obige Aktionen, die in der Shell des Manager Nodes ausgeführt worden sind (Abb. 15.44):

- Eine Text-Datei erstellen.
- Den Inhalt der Datei anzeigen.
- Ein Docker Secret aus dieser Datei erstellen.
- Die Liste der Docker Secrets anzeigen.

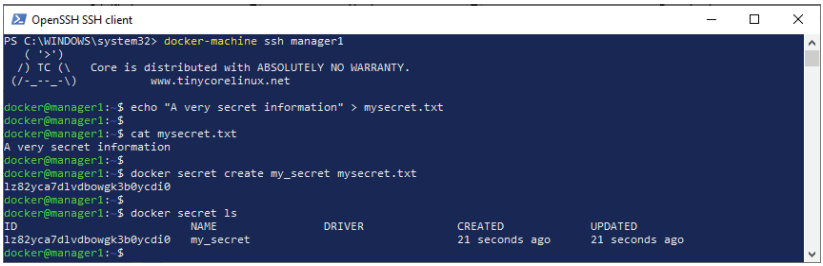



**Abb. 15.44** Docker Secret erstellen

Auch ein Secret kann man mit einem Inspect-Kommando untersuchen:

```
1  $ docker secret inspect [<OPTIONS>] <SECRET> [<SECRET>]
```

Hier der Aufruf des Inspect-Kommandos ohne Option (Abb. 15.45):

```
1  $ docker secret inspect my_secret
```

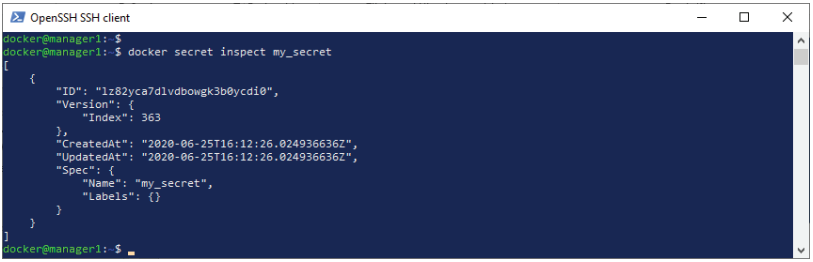



**Abb. 15.45** Docker Secret untersuchen

Jetzt sehen wir zum ersten Mal den Unterschied zu Docker Configs. Hier fehlt nämlich bei der Ausgabe das Element „`Data:`“, welches beim Kommando `docker config inspect` im Anschluss an das Element „`Labels:`“ angezeigt wird.

Auch bei Angabe der Option `--pretty` bekommt man die Daten eines Secrets nicht zu Gesicht (Abb. 15.46).

```
1  $ docker secret inspect --pretty my_secret
```

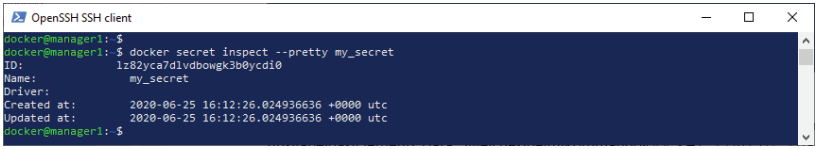



**Abb. 15.46** Docker Secret `inspect`-Kommando mit Option `--pretty`

Der Node der virtuellen Maschine `'worker1'` sollte immer noch den Manager-Status innehaben.

Wir wechseln wieder in die Kommando-Shell von `'worker1'`, um dort zu prüfen, ob man auch von dort auf das neu erstellte Secret-Zugriff hat. Geben Sie dort die Kommandos `docker secret ls` und `docker secret inspect pretty my_secret` ein (Abb. 15.47).

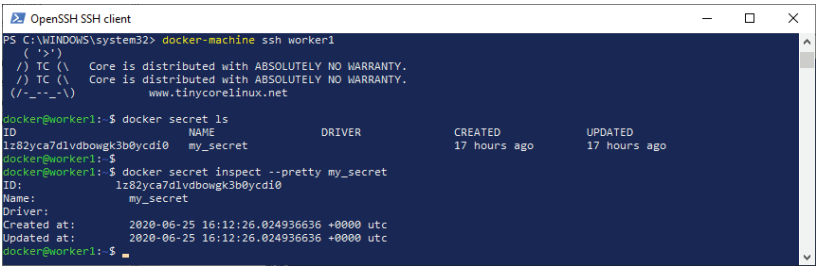



**Abb. 15.47** Zugriff auf ein Docker Secret aus einem anderen Manager Node

Sie haben das vermutlich schon erwartet. Es kann auf Secrets aus einem anderen Manager Node des Swarms zugegriffen werden und man kann auch aus jeder Manager Shell heraus neue Secrets für den zugehörigen Swarm anlegen.

Wie gesagt, aus einem Worker Node heraus, bei uns zum Beispiel worker2, funktionieren die gerade vorgeführten Befehle nicht. Docker reagiert hier mit einer Fehlermeldung:

```
1 Error response from daemon: This node is not a swarm manager.
2 Worker nodes can't be used to view or modify cluster state.
3 Please  run this command on a manager node or promote the
```

```
4  current node to a manager.
```

### 15.7.2 Docker Secrets an einen Service übergeben

Wie Sie ja mittlerweile schon erfahren haben, verwendet Docker bei Configs und Secrets das gleiche Konzept. Wie bei den Configs werden auch Secrets für einen Swarm erstellt, danach kann dann bei Bedarf den Services in einem Swarm der Zugriff auf benötigte Secrets gewährt werden.

Genau wie bei Konfigurationen werden auch Secrets beim Erstellen eines Service via Parameter übergeben.

Der Parameter für Secrets hat den Namen, welch eine Überraschung, `--secret` mit dem Namen des Secrets im Anschluss.

Als Beispiel erzeugen wir wieder einen `docker service` mit dem Namen `'my_redis'` mit drei Containern aus dem Image `'redis:alpine'`. Dieser Service erhält jetzt den Zugriff auf unser oben erstelltes Secret mit dem Namen `'my_secret'`.

```
1  $ docker service create --name my_redis --secret my_secret\
2   --replicas=3 redis:alpine
```

Falls der Service bereits existiert und läuft, dann müssen wir diesen nicht notwendigerweise entfernen und neu starten. In diesem Fall ist es einfacher, dem laufenden Service den Zugriff auf ein Secret mithilfe des Kommandos `docker service update` zu gewähren. Als Parameter muss dabei die Option `--secret-add <SECRET>` übergeben werden.

```
1  $ docker service update --secret-add my_secret my_redis
```

Nachdem wir das Secret an den Service `'my_redis'` übergeben haben, untersuchen wir wieder die Container, die zu Service `'my_redis'` gehören, und sehen nach, wo sich das zugewiesene Secret in den Containern befindet.

Führen wir im Container von `'my_redis'` auf dem Node `manager1` das Linux-Kommando `ls` aus (unsere `redis` Container basieren ja auf der ‚alpine'-Distribution von Linux) und lassen uns jetzt den Inhalt des

Verzeichnisses `/run/secrets` ausgeben. Dort sind standardmäßig die Mount Points abgelegt. Auch diese erhalten die Namen der zugehörigen Secrets.

```
$ docker container exec $(docker ps --filter name=my_redis -q)\
 ls -l /run/secrets
```

Als Mount Point wurde der Konfigurationsname hier im Verzeichnis `/run/secrets` des Containers angelegt. Auch diese Position ist bei Linux Containern Standard.

Bei Windows Containern werden Secrets in das Verzeichnis `'C:\ProgrammData\Docker\internal\secrets'` montiert. Die symbolischen Links auf diese Objekte befinden sich dabei per Default im Verzeichnis `,C:\ProgrammData\Docker\secrets‘`.

Lassen wir uns auch noch den Inhalt der Datei `'my_secret'` anzeigen.

```
$ docker container exec $(docker ps --filter name=my_redis -q)\
 cat /run/secrets/my_secret
```

Die vorgestellten Kommandos können Sie noch einmal bei ihrer Ausführung als Screenshot ansehen (Abb. 15.48). In diesem Beispiel wurde das Secret an den aktiv ausgeführten Service mithilfe des Kommandos `docker service update` übergeben.

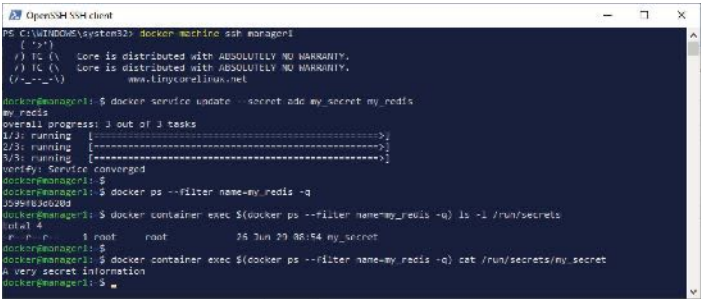



**Abb. 15.48** Docker Secrets an einen Service und dessen Container übergeben

Den Test, ob unser Secret in den Containern des Service, die auf den anderen Nodes laufen, auch noch verfügbar ist, wollen wir auch noch durchführen.

Wir wechseln noch einmal in die Shell des Nodes `'worker2'` und lassen uns von hier den Inhalt des Secrets `'my_secret'` ausgeben.

Auch hier ist das Secret vorhanden und hat den richtigen Wert als Inhalt (Abb. 15.49).

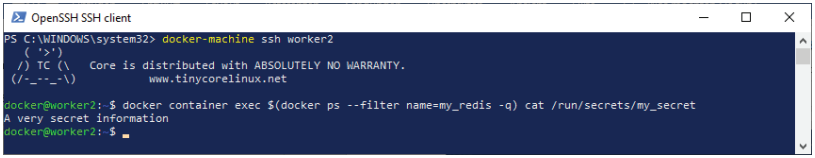



**Abb. 15.49** Docker Secrets auf verschiedenen Nodes eines Swarms

Versuchen wir jetzt das Secret zu löschen, obwohl es noch bei einem Service in Verwendung ist.

Hier heißt, das Kommando `docker secret rm`:

```
1  $ docker secret rm <CONFIG> [<CONFIG>]
```

Verbinden wir uns dazu erneut mit der Shell auf unserem Node `'manager1'`.

Für die Konfiguration `'my_secret'` sieht das Lösch-Kommando so aus:

```
1  $ docker secret rm my_secret
```

Es endet wieder, wie erwartet, mit der Fehlermeldung, dass das Secret `'my_secret'` noch vom Service `'my_redis'` benutzt wird und darum nicht gelöscht werden darf.

Wir entziehen dem Service den Zugriff auf das Secret wieder. Erst danach lässt es sich vom Swarm löschen. Das benötigte Kommando ist

wieder `docker service update`, diesmal mit der Option `secretrm`, gefolgt vom Namen des Secrets.

Beispiel:

```
1  $ docker service update --secret-rm my_secret my_redis
```

Jetzt löschen wir das Secret wieder (Abb. 15.50).

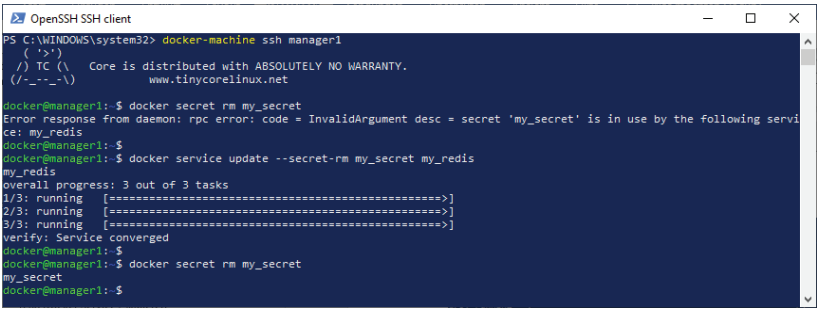



**Abb. 15.50** Ein Docker Secret wieder löschen

Zu guter Letzt entfernen wir auch noch den Service ‚`my_redis`', damit für das nächste Kapitel wieder alles aufgeräumt ist.

```
1  $ docker service rm my_redis
```

## 15.8 Einen Swarm auflösen

Ab und zu kann es vorkommen, dass man einen Docker Swarm komplett auflösen muss.

Damit das möglichst reibungslos abläuft, möchte ich Ihnen eine Auflistung der Aktionen in der richtigen Reihenfolge zur Hand geben:

1. Wechseln Sie in die Shell des Manager Nodes und stellen Sie dort sicher, dass alle Services beendet sind:

```
1  $ docker service ls
2  $ docker service rm <SERVICE> [<SERVICE> ...]
```

2. Wechseln Sie in die Shell eines jeden Worker Node und geben das Kommando ein, mit dem ein Node den Schwarm verlassen kann:

```
1 $ docker swarm leave
```

3. Wechseln Sie wieder in die Shell der Manager Nodes und geben das Kommando ein, mit dem ein Node den Schwarm verlassen kann. Geben Sie dabei die Option `--force` als Parameter an, um das Verlassen des Nodes zu erzwingen:

```
1 $ docker swarm leave --force
```

4. Falls die Nodes auf virtuellen Maschinen von Docker Machine angelegt wurden und diese werden nicht mehr benötigt, dann löschen Sie zuletzt die virtuellen Maschinen:

```
1 > docker-machine rm manager1 worker1 worker2
```

Kapitel 16

# Docker Stack

Mit zunehmender Anzahl der Services in einem Cluster wird es immer schwieriger, die Übersicht zu behalten, vor allem wenn dann auch noch über Netzwerke kommuniziert werden soll und gleichzeitig Secrets und Service Configs über die verschiedenen Services und deren Container korrekt verteilt werden müssen.

Hier kommt jetzt Docker Stack ins Spiel. Docker Stack befindet sich auf einer Abstraktionsebene über den Services und deren Abhängigkeiten. Das hört sich zunächst recht kompliziert an, aber es werden Ihnen viele Dinge bereits bekannt vorkommen. Einen Großteil der Elemente aus Docker Compose treffen wir bei Docker Stack wieder an. Das für Sie bereits bekannte YAML-Format von Compose wird weiterverwendet und um neue Elemente ergänzt, welche für die Arbeit mit Docker Swarm und das Deployment von Docker Services eingeführt worden sind.

So wie das auch bei Docker Compose der Fall ist, werden im Grunde genommen auch bei Docker Stack in einer YAML-Datei die Informationen eingetragen, die ansonsten, bei manueller Erstellung der Services, durch den Aufruf verschiedener Kommandos, wie zum Beispiel `docker service create` oder `docker secret create`, mit den nötigen Parametern von Hand eingegeben werden müssen.



## 16.1 Docker Stack in einer Single Node-Umgebung

Für die folgenden Beispiele zur Einführung in Docker Stack nutzen wir einen Single Node Swarm. Für die Kommandos und Konfigurationsdateien aus den Beispielen macht es keinen Unterschied, ob ein Swarm aus nur einem oder aus mehreren Nodes besteht. Falls Sie die Beispiele in einer Multi Node-Umgebung aus virtuellen Docker-Maschinen ausführen wollen, dann sollten Sie mit dem Linux Editor ***„vi“*** umgehen

können. Das ist der Editor, welcher auf der Kommando-Shell einer virtuellen Maschine zur Verfügung steht.

Starten Sie jetzt eine Windows PowerShell, wenn Sie auf einem Windows-Rechner arbeiten, oder eine andere Kommando-Shell Ihrer Wahl und initialisieren Sie den Swarm Mode:

```
1  > docker swarm init
```

Damit wir sehen, ob unser Rechner als Node angelegt wurde, lassen wir uns die Liste der Nodes ausgeben:

```
1  > docker node ls
```

Wenn alles funktioniert hat, dann wird Ihr Arbeitsrechner (z.B. docker-desktop) als Manager Node des Swarms angezeigt.

### 16.1.1 Ein erster ganz einfacher Stack

Wir wollen uns langsam und Schritt für Schritt an das Thema Docker Stack herantasten. Darum beginnen wir mit einem ganz simplen Beispiel.

Dazu erstellen wir uns mithilfe von Docker Stack einen Service, der aus einem einzigen NGINX Container besteht.

Legen Sie für das Beispiel ein neues Verzeichnis mit dem Namen `'DockerStack'` unter Ihrem Benutzerverzeichnis an.

```
1  <USER_DIR>\DockerStack
```

Um für Docker Stack die Eigenschaften eines Swarm Services zu bestimmen, benötigen wir, wie schon bei Docker Compose, eine Konfigurationsdatei im YAML-Format. Diese Datei soll den Dateinamen `'docker-stack.yaml'` erhalten. Der Inhalt dieser Datei wird erst einmal so erstellt, wie wir das schon im Kapitel für Docker Compose kennengelernt haben.

```
Datei 'docker-stack.yaml'
version: "3.7"
services:
    my_web:
        image: nginx:1.17.7
        ports:
            - "8080:80"
```

ACHTUNG: Die Version der Compose-Datei muss für Docker Stack größer oder gleich der Version „3.0“ sein. Wir verwenden hier Version „3.7“.

Unser Service erhält wieder den Namen `'my_web'` und soll aus dem Docker Hub Image `nginx` mit der Version 1.17.7 erstellt werden.

Der interne Port 80 soll auf dem externen Port 8080 veröffentlicht werden.

Bis hierher ist alles wie gehabt. Jetzt folgt der Unterschied zu Docker Compose. Es kommt für die Arbeit mit Docker Stack ein neues Kommando zum Einsatz, das Kommando `docker stack deploy`. Damit erzeugen wir für einen Stack alle im YAML File definierten Services:

So sieht die Syntax aus:

```
docker stack deploy [<OPTIONS>] STACK
```



Als wichtigste Option verwenden wir dabei

```
--compose-file oder -c
```

Damit gibt man Pfad und Dateinamen der zu verwendenden Compose-Datei an. Nebenbei bemerkt: Es kann auch hier ein Minus-Zeichen (-) angegeben werden, um STDIN anstelle einer Datei als Eingabe-Stream zu verwenden.

Wechseln Sie in Ihrer Kommando-Shell in das Verzeichnis, in dem sich die Compose-Datei befindet, und geben Sie dort das folgende Kommando ein:

```
> docker stack deploy --compose-file docker-stack.yaml my_stack
```

Bei diesem Beispiel erzeugen wir den Service gemäß den Einträgen in der Compose-Datei `'docker-stack.yaml'`. Der Stack erhält den Namen `'my_stack'`.

Auch ein `ls`-Kommando gibt es für den Docker Stack. Damit lässt man sich die Liste der verfügbaren Stacks ausgeben:

```
> docker stack ls
```

Wir lassen uns auch noch die Liste aller Services anzeigen, die zu einem Stack gehören.

Syntax:

```
docker stack services [<OPTIONS>] STACK
```

Als wichtigste Option verwenden wir dabei

```
--format
```

, um die Ausgabe durch Angabe eines GO-Templates zu formatieren.

Mit diesem Kommando lassen wir uns die Liste der Services von `'my_stack'` anzeigen:

```
> docker stack services my_stack
```

Natürlich wollen wir uns auch alle Tasks, die zu einem Stack gehören, mit ihren Attributen auflisten lassen.

Die Syntax für das zugehörige `docker stack`-Kommando:

```
docker stack ps [<OPTIONS>] STACK
```

Auch hier gibt es, unter anderem, die Option

```
--format
```

zur formatierten Ausgabe mittels GO-Template.

Wir geben also für unseren Stack das Kommando wie folgt ein:

```
> docker stack ps my_stack
```

Der nächste Screenshot zeigt obige Kommandos in Aktion, wenn sie in der PowerShell ausgeführt werden (Abb. 16.1):

- Deployment des Stacks.
- Die Liste der Stacks anzeigen.
- Die Liste der Services für den Stack `'my_stack'` anzeigen.
- Die Liste der Tasks für den Stack `'my_stack'` anzeigen.

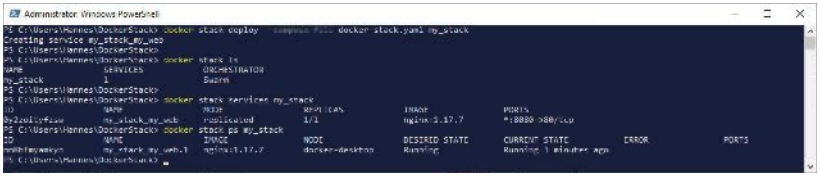



**Abb. 16.1** Einen einfachen Docker Stack einrichten

Es fehlt jetzt nur noch ein letztes Kommando zur Stack-Verwaltung, nämlich das Kommando, um einen Stack wieder zu entfernen und dabei alle laufenden Services mit ihren Tasks zu beenden. Wie bei den Docker-Kommandos üblich, heißt das Kommando auch hier `rm`. Die vollständige Syntax für dieses Kommando:

```
> docker stack rm <STACK> [<STACK> ...]
```

So entfernen wir den Stack mit dem Namen `'my_stack'` und prüfen danach, ob das Kommando erfolgreich ausgeführt wurde (Abb. 16.2):

```
> docker stack rm my_stack
> docker stack ls
```


**Abb. 16.2** Einen Docker Stack entfernen

### 16.1.2 Stack Service mit mehreren Replikaten

Bis jetzt sehen wir noch keinen Unterschied von Docker Stack im Vergleich zu Docker Compose, abgesehen davon, dass jetzt der Swarm Mode initialisiert sein muss und andere Kommandos ausgeführt werden.

Aber im Swarm Mode haben wir die Möglichkeit, bei den Services mehrere Replikate der Container zu starten. Das haben wir bis jetzt beim Start des Kommandos `docker service create` durch Angabe der Option `--replicas=<ANZAHL>` erledigt.

Bei Docker Stack wird auch diese Information in der Docker-Compose-Datei angegeben.

Um den Service `'my_web'` mit drei Replikaten zu starten, fügen wir der Datei `'docker-stack.yaml'`, wie im Beispiel unten gezeigt, eine neue Sektion mit dem Namen `'deploy:'` zu. Darunter tragen wir den Key `'replicas:'` mit dem Wert `'3'` ein.

```
Datei 'docker-stack.yaml'
version: "3.7"
services:
    my_web:
        image: nginx:1.17.7
        ports:
            - "8080:80"
        deploy:
            replicas: 3
```

Damit haben wir erreicht, dass bei Ausführung von `docker stack deploy` mit dieser Compose-Datei der Service `'my_web'` mit drei Container-Instanzen erstellt wird.

```
> docker stack deploy --compose-file docker-stack.yaml my_stack
```

Nach Ausführung des Kommandos sehen wir uns erneut die Liste der Stacks an:

```
> docker stack ls
```

und die Liste der Services von `'my_stack'`:

```
> docker stack services my_stack
```

Zuletzt wird die Liste der Tasks von `'my_stack'` ausgegeben:

```
> docker stack ps my_stack
```

Noch ein Screenshot des PowerShell-Fensters mit den oben angegebenen Kommandos in Aktion (Abb. 16.3):

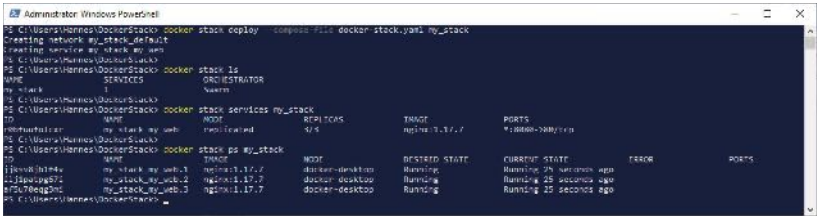



**Abb. 16.3** Eine Docker Stack mit Service-Replikaten

Bei Aufruf http://localhost:8080/ im Webbrowser erscheint wieder die bekannte NGINX Standard-Seite „Welcome to NGINX“.

Bevor wir mit dem nächsten Thema weitermachen, löschen wir den Stack wieder:

```
> docker stack rm my_stack
```

### 16.1.3 Configs mit Docker Stack verwalten

Ein weiteres Feature, das wir für Docker im Swarm Mode kennengelernt haben, stellt die Anwendung von Configs dar. Damit wird es möglich, unkritische Daten bzw. Informationen an die Container eines Service

zu übergeben. Bisher haben wir eine Konfiguration mit dem Kommando `docker config create` erstellt und beim Anlegen eines Service mit dem Kommando `docker service create` über den Parameter `--config` an den Service übergeben und damit den Containern-Instanzen des Service den Zugriff auf die Konfiguration ermöglicht.

Mit Docker Stack vereinfacht sich auch der Einsatz von Configs, weil auch diese über Docker-Compose-Dateien verwaltet werden.

Um das praktisch vorzuführen, wird das NGINX-Beispiel erweitert. Wir wollen die Webseite „Welcome to NGINX“, die vom NGINX Container standardmäßig bereitgestellt wir, wieder durch eine eigene Webseite ersetzen. Wir erstellen die Webseite nicht noch einmal neu, sondern kopieren diejenige aus unserem früheren Beispiel `'DoubleService'`. Wenn Sie für das Übungsverzeichnis den Vorschlag aus diesem Buch übernommen haben, dann liegt die damals angelegte HTML-Datei in folgendem Verzeichnis:

```
1  <USER_DIR>\DoubleService\html
```

Dieses Verzeichnis mit der enthaltenen Datei `'index.html'` kopieren wir einfach in das Übungsverzeichnis für das Docker Stack-Beispiel:

```
1  <USER_DIR>\DockerStack
```

Der Inhalt der Datei `'index.html'` soll jetzt unsere Konfigurationsdaten liefern.

Um das mit Docker Stack zu erledigen, muss man die Compose-Datei `'docker-stack.yaml'` erweitern.

Da muss als Erstes die Konfiguration erstellt werden. Am Ende des folgenden Beispiel-Listings für `'docker-stack.yaml'` befindet sich eine neue Sektion `configs`. Sie muss für die oberste Ebene angelegt werden, also ohne Einrückung. Darunter, eine Ebene tiefer, folgen dann die Namen der Configs. Unter jedem Namen wird dann, wiederum eine Ebene tiefer, die Quelle der Konfigurationsdaten angegeben.

Das entspricht der Erstellung einer Konfiguration mit dem Kommando `docker config create`. Falls sie eine Config angeben wollen, die mit diesem Kommando erstellt wurde und damit bereits existiert, dann kann hier auch der Name dieser Config eingetragen werden. Anstelle des Dateinamens folgt eine Key-Value-Angabe `external: true`.

Um eine so erstellte Config einem Service zuzuweisen, wird unter der Sektion dieses Service wieder eine Sektion mit dem Namen `configs:` eingefügt. Darunter, eine Ebene weiter eingerückt, gibt man dann eine oder mehrere Config-Einträge im Listenformat (mit führendem Minus-Zeichen) an.

```
Datei 'docker-stack.yaml'
version: "3.7"
services:

    my_web:
        image: nginx:1.17.7
        ports:
            - "8080:80"
        networks:
            - test_net
        deploy:
            replicas: 3
        configs:
            - source: html.conf
              target: /usr/share/nginx/html/index.html

networks:
    test_net:

configs:
    html.conf:
        file: ./html/index.html
```

ANMERKUNG: In obigem Beispiel-Listing wird auch noch ein Netzwerk definiert und für dieses der Name `test_net` vergeben. Lässt man das weg, wird trotzdem ein Netzwerk mit dem Namen `default` erzeugt.

Erzeugen wir den Stack neu und testen dann die Änderungen:

```
> docker stack deploy --compose-file docker-stack.yaml my_stack
```

Der Screenshot zeigt die folgenden Aktionen (Abb. 16.4):

- Erstellen des Stacks `'my_stack'`.
- Anzeige der Liste der Stacks.
- Anzeige der Liste der Services von `'my_stack'`.
- Anzeige der Tasks von `'my_stack'`.

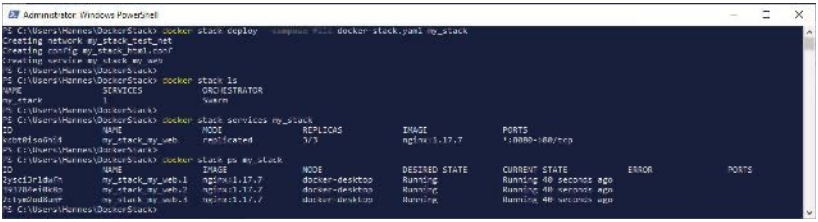



**Abb. 16.4** Docker Stack mit Configs

Nach Aufruf des Kommandos `docker stack deploy` wird unter anderem die Meldung „**Creating config my_stack_html.conf**“ ausgegeben. Sie können hier sehen, dass der Name einer Config von Docker durch die Kombination des Stack-Namens und dem Config-Namen aus der Compose-Datei generiert wird.

Bei Aufruf http://localhost:8080/ im Webbrowser erscheint die von uns erstellte Webseite (Abb. 16.5).


**Abb. 16.5** Eine eigene NGINX-Webseite mit Docker Stack

Wir beenden dieses Beispiel und räumen wieder auf:

```
> docker stack rm my_stack
```

### 16.1.4 Secrets im Stack verwalten

Da die Konzepte von Docker Configs und Secrets sehr ähnlich sind, ist jetzt der Einsatz von Secrets mit Docker Stack recht einfach erklärt.

Um die Anwendung von Secrets zu demonstrieren, erstellen wir eine SQL-Datenbank-Applikation mit MariaDB und phpMyAdmin. Die Passwörter für den Root Account der Datenbank und den Benutzer verwalten wir jetzt aber mithilfe von Docker Secrets.

Wir verwenden für das Beispiel wieder ein neues Verzeichnis unterhalb des Benutzerverzeichnisses. Das soll den Namen `'DockerStackDB'` erhalten:

```
<USER_DIR>\DockerStackDB
```

Wir erstellen darunter noch ein Unterverzeichnis mit dem Namen `'secrets'`:

```
<USER_DIR>\DockerStackDB\secrets
```

Dort legen wir zwei Text-Dateien an, die unsere Passwörter für die MariaDB-Datenbank bereithalten sollen. Eine erhält den Namen `'mysql_root_password.txt'` und die andere `'mysql_password.txt'`. Die dort eigetragenen Passwörter bleiben natürlich Ihr Geheimnis (z.B. ‚topsecret').

```
<USER_DIR>\DockerStackDB\secrets\mysql_root_password.txt
<USER_DIR>\DockerStackDB\secrets\mysql_password.txt
```

Die Docker-Compose-Datei für dieses Beispiel nennen wir `'docker-stack-db.yaml'`. Sie besteht im Kern aus den Einträgen, wie wir sie schon aus dem Docker-Compose-Beispiel zum Erstellen einer Datenbank-Anwendung mit MariaDB kennen. Erweitert wird sie um die Elemente, die für die Verwaltung von Secrets unter Docker Stack benötigt werden.

Am Ende der Compose-Datei stehen die Einträge, welche für die Erstellung der Secrets zuständig sind. Die Sektion beginnt mit dem Schlüsselwort `secrets:` und sie muss ebenfalls in der obersten Ebene liegen. Sie beginnt also in der ersten Spalte. Darunter, eine Ebene tiefer, folgen dann die Namen der Secrets. Unter jedem Namen wird dann, wiederum eine Ebene tiefer, die Quelle der Daten für das zugehörige Secret angegeben.

Das entspricht jetzt der Erstellung einer Secret mit dem Kommando `docker secret create`. Auch bereits existierende, außerhalb der Compose-Datei erstellte Secrets, können hier mit Ihrem Namen angegeben und so wie auch externe Configs durch den Eintrag `external: true`. gekennzeichnet werden.

Um die so erstellten Secrets den Services zuzuweisen, wird unter der Sektion des entsprechenden Service wieder eine Sektion mit dem Namen `secrets:` eingefügt. Darunter, eine Ebene weiter eingerückt, gibt man dann einen oder mehrere Secret-Namen im Listenformat (mit führendem Minus-Zeichen) an.

Vielleicht fragen Sie sich, was in der Beispiel-Compose-Datei der Eintrag

```
restart_policy:
    condition: any
```

in der deploy-Sektion zu bedeuten hat? Damit müssen wir für den Swarm Mode die `'restart: always'`-Anweisung von Docker-Compose ersetzen:

```
Datei 'docker-stack-db.yaml'
version: "3.7"
services:
    database:
        image: mariadb:latest
        volumes:
            - mariadb_data:/var/lib/mysql
        ports:
            - "3306:3306"
        environment:
            MYSQL_ROOT_PASSWORD_FILE: /run/secrets/mysql_root_
password
            MYSQL_DATABASE: telefon
            MYSQL_USER: user
            MYSQL_PASSWORD_FILE: /run/secrets/mysql_password
        deploy:
            restart_policy:
                condition: any
            replicas: 1
        networks:
            - db_net
        secrets:
            - mysql_root_password
            - mysql_password

    phpmyadmin:
        image: phpmyadmin/phpmyadmin
        ports:
            - "8080:80"

        environment:
            PMA_HOST: database
            PMA_PORT: 3306
            MYSQL_ROOT_PASSWORD_FILE: /run/secrets/mysql_root_
password
        depends_on:
            - database
```

```
        deploy:
            restart_policy:
                condition: any
            replicas: 2
        networks:
            - db_net

        secrets:
            - mysql_root_password

networks:
    db_net:

volumes:
    mariadb_data:

secrets:
    mysql_root_password:
        file: ./secrets/mysql_root_password.txt
    mysql_password:
        file: ./secrets/mysql_password.txt

```

Starten Sie erneut eine Shell und wechseln Sie dort in das Verzeichnis, in dem sich diese YAML-Datei befindet (`'<USER_DIR>\Docker StackDB'`). Wir erzeugen mit den Einträgen dieser Datei unseren neuen Stack durch Eingabe des nachfolgenden Docker-Kommandos:

```
> docker stack deploy --compose-file docker-stack-db.yaml '
my_db_stack
```

Anschließend kontrollieren wir das Ergebnis der Kommandos, indem wir uns die Liste der Stacks anzeigen lassen:

```
> docker stack ls
```

Nun geben wir noch die Liste der Services von `'my_stack'` aus:

```
> docker stack services my_db_stack
```

Zuletzt untersuchen wir noch einmal die Liste der Tasks von `my_stack`:

```
> docker stack ps my_db_stack
```

Hier wieder ein Screenshot von einer PowerShell, der die oben aufgeführten Kommandos in Aktion zeigt (Abb. 16.6):

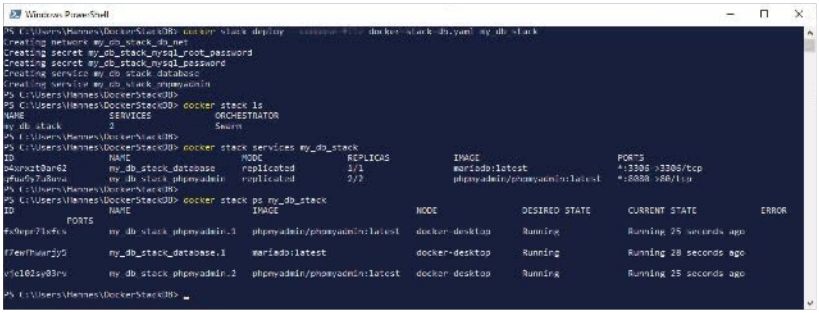



**Abb. 16.6** Ein Docker Stack mit Secrets

Zur Überprüfung starten Sie einen Internet-Browser und geben noch einmal `'http://localhost'` mit der Portnummer 8080 als URI an. Wenn alles geklappt hat, sehen Sie wieder die Anmeldeseite von phpMyAdmin wie im Kapitel 14.1 (siehe Abb. 14.1).

Kapitel 17

# Kubernetes

In diesem Kapitel stellen wir Ihnen ein Produkt vor, welches als Alternative zu Docker Swarm eingesetzt werden kann und das mittlerweile in aller Munde ist. Von vielen Experten wird gerade in der nahen Zukunft ein flutartiger Anstieg für den Einsatz von Kubernetes bei webbasierten Anwendung erwartet.

Was ist Kubernetes also eigentlich?

Der Name Kubernetes ist die englische Schreibweise des griechischen Wortes κουμπερνέτες. was so viel bedeutet wie Steuermann oder Rudermann. Diese Bedeutung spiegelt sich auch im Logo von Kubernetes wider, einem Schiffsruder.

Kubernetes ist, so wie Docker Swarm auch, ein weiteres Tool zur Orchestrierung von containerbasierten Anwendungen. Es handelt sich hier um ein System zur Automatisierung, Bereitstellung, Skalierung und Verwaltung von großen Container-Applikationen.



Mit Kubernetes ist es möglich, hochverfügbare und ausfallsichere Lösung zu realisieren.

Ein wichtiger Vorteil von Kubernetes gegenüber Docker Swarm ist, dass beim Einsatz von Kubernetes eine einfache Portierung von Container-

Anwendungen zwischen verschiedenen Cloud-Diensten möglich ist, vorausgesetzt diese sind kompatibel zu Kubernetes.

Anders als Docker Swarm ist Kubernetes aber kein Bestandteil von Docker. Allerdings wird von Docker inzwischen die Bereitstellung von Containern über Kubernetes unterstützt.

Bei der Verwaltung von Containern ist man aber bei Kubernetes nicht auf Docker beschränkt. Es können damit neben Docker Containern auch andere Container-Produkte bereitgestellt und orchestriert werden.

Woher kommt Kubernetes?

Kubernetes wurde von Google entwickelt. Seine Entwicklung wurde schon 2014 von Google angekündigt.

Die Entwicklung von Kubernetes baut auf den Erfahrungen auf, die bei Google in vielen Jahren der Entwicklung von Web-Anwendungen mit Containern gesammelt wurden. Kubernetes wurde übrigens, so wie viele andere Anwendungen im Cloud-Umfeld auch, in der Programmiersprache GO geschrieben.

Die erste Version 1.0 wurde dann am 21. Juli 2015 von Google veröffentlicht. Zu dieser Zeit wurde auch die „Cloud Native Computing Foundation“ als Unterprojekt der „Linux Foundation“ gegründet. Diese Organisation verfolgt das Ziel, Container-Technologien voranzutreiben und zu unterstützen sowie die Weiterentwicklung durch die verschiedenen beteiligten Technologie-Unternehmen einander anzugleichen.

Google hat Kubernetes an diese Stiftung gespendet und bezeichnet Kubernetes selbst als GIFEE-Projekt (Google Infrastructure for Everybody Else).

Herzlich Willkommen bei Kubernetes und viel Spaß beim Kennenlernen!

## 17.1 Das Zusammenspiel von Docker und Kubernetes

Bei Kubernetes und Docker handelt es um komplementäre Technologien, die sich perfekt ergänzen.

Kubernetes ist ein Tool das lediglich die Orchestrierung von Containern übernehmen kann. Es kann aber selbst keine Container erstellen.

Bei der Entwicklung einer Anwendung entwickelt man den Code dazu in einer Programmiersprache seiner Wahl. Dann setzt man Docker ein, um die Anwendung in Container zu packen und zu testen. Es können allerdings auch andere Tools zur Erstellung von Containern genutzt werden.

Die fertigen Container bindet man anschließend in Kubernetes ein. Die Orchestrierungsfunktion von Kubernetes übernimmt später im Betrieb die Bereitstellung, Verwaltung und Skalierung der Container.

Beim Zusammenspiel von Docker und Kubernetes übernimmt Docker die low-level-Aufgaben wie zum Beispiel das Starten und das Stoppen von Container Applikationen. Kubernetes ist eine Abstraktionsebene höher angesiedelt und entscheidet auf dieser Ebene, welche Container auf welchen Nodes ausgeführt werden, wann die Skalierung von Services erhöht oder reduziert wird oder wann Updates ausgeführt werden.

## 17.2 Docker Swarm und Kubernetes: eine Gegenüberstellung

Docker und Kubernetes kommen bestens miteinander aus und sind problemlos kombinierbar.

Anders sieht es bei Docker Swarm aus. Hier kann man durchaus von Konkurrenz sprechen. So lassen sich Docker Swarm und Kubernetes nicht miteinander kombinieren.

Man muss sich somit entscheiden, ob man auf Swarm setzt, welches einen integralen Bestandteil der Docker-Distributionen darstellt oder

ob man sich für Kubernetes entscheidet, welches sich immer größerer Beliebtheit erfreut. Im Moment sieht es so aus, als ob Kubernetes bei diesem Rennen die Nase immer weiter vorne hat.

Beide Tools haben die Aufgabe, Container effizient zu verwalten, den Einsatz der benötigten Ressourcen zu optimieren und die Komponenten von Systemen je nach Bedarf optimal zu skalieren.

Was die Verfügbarkeit der Services angeht, kann man beide Systeme in etwa als gleichwertig einstufen.

Ein Vorteil von Swarm ist sicher, dass es nicht extra installiert werden muss. Kubernetes muss dagegen parallel zu Docker installiert werden. Das ist aber nicht besonders aufwendig.

Auch was die Themen Skalierbarkeit und Load Balancing angeht, wird Docker Swarm etwas besser bewertet als Kubernetes.

Kubernetes bietet dem Anwender im Gegenzug eine sehr komfortable und übersichtliche Benutzerschnittstelle, mit deren Hilfe viele Aufgaben recht einfach zu erledigen sind.

Kubernetes bietet auch mehr Funktionen zum Monitoring und Logging von Applikationen und deren Containern während des Betriebs.

## 17.3 Kubernetes-Grundlagen

In diesem Kapitel lernen Sie die wichtigsten Bestandteile von Kubernetes kennen, die man benötigt, um Cluster anzulegen und Applikationen darüber bereitzustellen. Damit soll Ihnen zum Einstieg in das Thema eine Übersicht über die wichtigsten Konzepte von Kubernetes vorgestellt werden.

### 17.3.1 Das Kubernetes-Cluster

Beginnen wir mit einem Begriff, den Sie schon aus der Welt von Docker Swarm kennen, dem Cluster.

Kubernetes-Clustern setzen sich aus mehreren Linux Hosts zusammen. Diese können sowohl auf virtuellen Maschinen laufen als auch auf physikalischen Computern. Sie können aber auch als virtuelle Instanzen in einer Cloud realisiert worden sein.

Wie schon bei Docker Swarm gibt es auch bei Kubernetes-Clustern zwei unterschiedliche Rollen:

Die Rolle des *Managers*, die wir bei Docker Swarm kennengelernt haben, bezeichnet man bei Kubernetes als „Master".

Die Rolle des *Workers* aus Docker Swarm nennt man bei Kubernetes einfach „Node".

Betrachten wir zunächst die Aufgaben dieser beiden Cluster-Rollen.

#### *17.3.1.1 Master*

So wie ein *Swarm Manager* bildet ein *Kubernetes Master* die Kontrollebene eines Clusters und ist für dessen Organisation und Koordination verantwortlich.

Zu den Aufgaben eines Kubernetes Master gehören unter anderem die Kontrolle des gewünschten Soll-Status einer Applikation (z.B. Anzahl der laufenden Pod-Instanzen), die Skalierung von Applikationen und die Bereitstellung neuer Updates für ein Cluster.

Werfen wir einen Blick auf die Dienste, aus denen ein Kubernetes Master zusammengesetzt ist (Abb. 17.1.)

- Der API-Server
- Der Cluster Store

- Der Control Manager
- Der Scheduler
- Der Cloud Control Manager

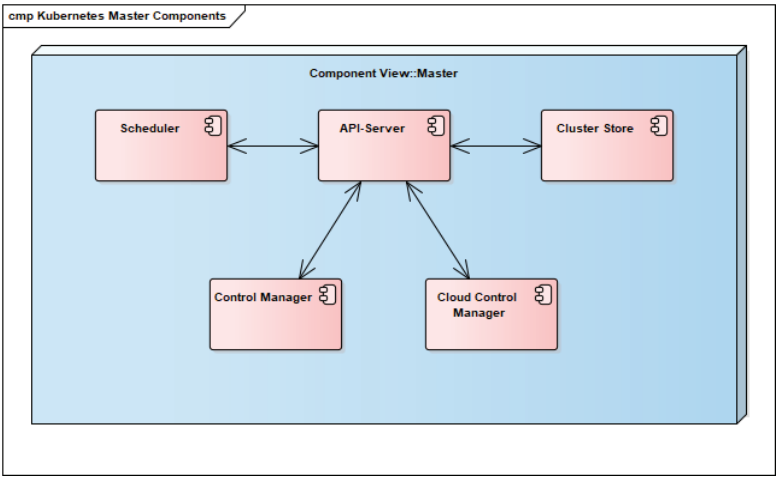



**Abb. 17.1** Komponenten eines Kubernetes Master

*API-Server*: Er stellt die zentrale Steuereinheit eines Kubernetes Clusters dar. Die gesamte Kommunikation zwischen allen Elementen des Clusters wird über ihn abgewickelt. Dabei benutzen alle Beteiligten die gleiche Schnittstelle. Es handelt sich um ein REST-konformes Protokoll, bei dem Konfigurationsdaten im YAML-Format, sie werden auch als Manifeste bezeichnet, über HTTPS ausgetauscht werden. Zu Wiederholung: REST steht für Representational State Transfer und repräsentiert ein Programmierparadigma für verteilte Systeme, insbesondere für Webservices. Zu den bekanntesten REST-konformen Protokollen gehören unter anderem die Protokolle HTTP und HTTPS.

*Cluster Store*: Der Cluster Store ist für die Speicherung der gesamten Konfiguration eines Clusters verantwortlich. Hier wird außerdem noch der aktuelle Status eines Clusters abgelegt und verwaltet. Die Informationen werden in einer verteilten Datenbank gehalten. Diese Datenbank basiert auf **etcd**, einem Open-Source-System, das kritische Daten hierarchisch als ‚key-value'-Paare ablegt. Diese Datenbank baut eben-

falls auf dem „Raft Consensus Algorithmus“ auf, den wir schon im Umfeld von Docker Swarm kurz vorgestellt haben.

*Control Manager*: Der Control Manager implementiert all die Schleifen, die als Hintergrund-Prozesse einen Cluster überwachen und auf auftretende Ereignisse reagieren. Er kontrolliert damit die anderen Controller innerhalb der Cloud wie zum Beispiel Node Controller, Endpoints Controller oder Replicaset Controller.

*Der Scheduler*: Ähnlich wie bei Betriebssystemen (im Grunde genommen ist Kubernetes ja so etwas wie ein Betriebssystem für Cloud-Computing) hat der Scheduler die Aufgabe, neue Tasks den Nodes zuzuweisen. Dabei überprüft der Scheduler den Status der Nodes. Unter anderem wird geprüft, ob ein Task über ausreichende Ressourcen verfügt oder ob ein Node über das Netzwerk erreichbar ist. Der Scheduler stellt damit sicher, dass nur solche Nodes einen Task zur Ausführung erhalten, die dazu auch in der Lage sind.

*Cloud Control Manager*: Falls ein Cluster innerhalb einer Cloud-Plattform, wie beispielsweise AWS, Azure, der IBM Cloud oder Google GCP, ausgeführt wird, dann aktiviert der Master des Clusters seinen *Cloud Control Manager*. Der ist verantwortlich für die reibungslose Einbindung des Clusters in die Infrastruktur des genutzten Cloud Hosting Service.

#### *17.3.1.2 Node*



Nodes sind die „Arbeitstiere“ in einem Cluster und für die eigentliche Funktion der Applikation verantwortlich.

Wie bereits angedeutet sind sie mit den Worker Nodes von Docker Swarm vergleichbar.

Kubernetes Nodes enthalten aber drei besondere Komponenten (Abb. 17.2):

- Das Kubelet

- Die Container Runtime
- Der Kube-Proxy

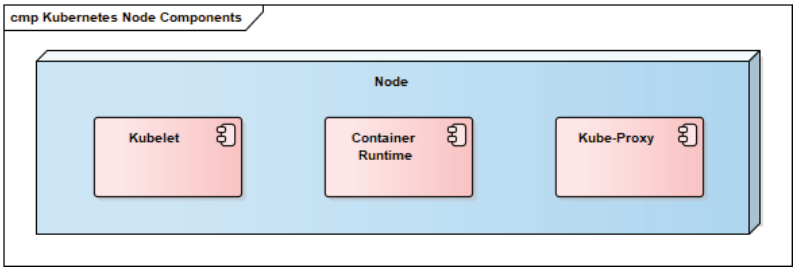



**Abb. 17.2** Komponenten eines Kubernetes Nodes

*Kubelet*: Übernimmt bei Kubernetes die Funktion eines Agenten von Kubernetes und muss auf jedem Node eines Clusters laufen. Zum einen ist die Kubelet-Komponente verantwortlich für die Registrierung seines Nodes im Cluster. Dabei werden die Ressourcen des Clusters (Speicher, Prozessor, ...) einem übergeordneten Cluster Pool zugeordnet. Die Hauptaufgabe eines Kubelet ist es, auf die Zuweisung neuer Tasks vom API-Server zu warten. Jede Anweisung, die ein Kubelet erhält, wird zunächst ausgeführt und das Ergebnis der Ausführung wird an die Kontrollebene zurückgemeldet. Ein Kubelet trifft aber selbst keine Entscheidungen. Wenn zum Beispiel eine Anweisung nicht oder nur fehlerhaft ausgeführt werden konnte, dann wird das lediglich zurückgemeldet. Die Kontrollschicht entscheidet letztendlich, was danach zu tun ist.

*Container Runtime*: Sie stellt die Laufzeitumgebung für Container bereit. Die Kubernetes Nodes benötigen die Container Runtime zur Durchführung von containerspezifischen Aufgaben, wie zum Beispiel dem Laden von Images oder dem Starten und Stoppen von Containern.

*Kube-Proxy*: Die dritte Komponente, die auf jedem Node eines Kubernetes Clusters aktiv ist. Er Ist verantwortlich für die lokale Netzwerkkommunikation im Cluster. Dabei weist er den Nodes eindeutige IP-Adressen zu. Zusätzlich übernimmt die Proxy-Komponente die Organisation für das Routing und Load-Balancing des Datenverkehrs im Netzwerk eines Clusters.

### 17.3.2 Das Domain-Name-System eines Kubernetes-Clusters

Ein DNS Service verbindet auch in Kubernetes-Clustern die IP-Adressen innerhalb des Clusters mit Domänen-Namen.

Für ein Cluster wird von Kubernetes automatsch ein DNS Service instanziiert, konfiguriert und gestartet.

Auch wenn neue Services erstellt, verschoben oder gelöscht werden, aktualisiert Kubernetes die zugehörigen Informationen im DNS Service des Clusters automatisch.

Der DNS Service selbst hat eine fixe IP-Adresse. Diese ist allen Komponenten des Clusters bekannt. Damit kann jeder Service über seinen Namen adressiert und mit ihm über das Cluster Netzwerk kommuniziert werden.

Damit gestaltet sich die Kommunikation der Dienste und Anwendungen in einem Cluster über das Netzwerk recht einfach.

### 17.3.3 Pods

Pods bilden die kleinste Einheit von Kubernetes.

Ein Pod ist der englische Begriff für eine Delfinschule, als Pod of Whales bezeichnet man eine Walschule, also einen Schwarm von Walen.



Das Logo für Docker ist ein Wal. Darum hat man sich wohl entschlossen für eine Gruppe von Containern den Begriff Pod zu verwenden.

Docker Container, wie auch Container aus anderen Systemen, können nicht direkt in einem Node ausgeführt werden. Dafür ist so etwas wie ein Adapter nötig. Diese Adapterfunktion erfüllen Pods. Sie bieten den Kubernetes Nodes eine einheitliche Schnittstelle zur Kontrolle von Containern unterschiedlicher Anbieter.

Ein Pod kann einen oder mehrere Container aufnehmen. Ein Pod stellt dafür ein sogenanntes ‚Shared Execution Environment' bereit. Das bedeutet, bestimmte Ressourcen stehen allen Containern zur Verfügung, die im selben Pod ausgeführt werden. Zu diesen Ressourcen gehören unter anderem IP-Adressen, Ports, der Host-Name, Routing Tabellen, Speicher oder Volumes.

Wenn Services mithilfe von Kubernetes skaliert werden sollen, dann wird die Anzahl der Pod-Instanzen verändert. Skalierung wird bei Kubernetes nicht durch die Änderung der Container-Instanzen innerhalb eines Pods realisiert.

### 17.3.4 Deployment

Im Allgemeinen versteht man unter dem Begriff Deployment die automatisierte Verteilung, Installation, Konfiguration und Wartung von Software auf mehreren Computersystemen.

Unter Kubernetes ist ein Deployment ein Objekt mit dessen Hilfe ein Set von identischen Pods verwaltet wird. Dabei wird jede Applikation in einem Cluster mithilfe von Deployment-Objekten verteilt, skaliert und auch gewartet.

Die Eigenschaften eines Deployment-Objekts werden mithilfe einer Datei beschrieben, welche im YAML-Format erstellt wird.

Ein Deployment dient also dazu, die Instanzen von skalierten Pods in einem Cluster zu steuern, und spielt dabei eine ähnliche Rolle wie Docker-Stack bei der Verwaltung von Container-Instanzen in einem Swarm.

### 17.3.5 Kubernetes Services

Der Begriff Service hat bei Kubernetes eine ganz andere Bedeutung, als wir es von Docker Swarm her kennen: Dort bezeichnet man einen

oder mehrere Container mit der gleichen Konfiguration, die im Swarm Mode von Docker ausgeführt werden, als Service.

Bei Kubernetes bieten Services den Client-Applikationen eine einheitliche Schnittstelle zu Kubernetes Pods innerhalb eines Deployment.

Warum ist das nötig?

Durch ein Deployment wird festgelegt, wie viele Instanzen eines Pods für eine Applikation aktiv sein sollen. Bei Fehlschlägen während der Ausführung werden Pods beendet und komplett neue Pod Instanzen werden als Ersatz gestartet. Diese neuen Instanzen erhalten dabei auch neue IP-Adressen als Schnittstelle. Auch wenn die Anzahl der Pods durch ein Deployment verändert wird, die Anzahl der Pods erhöht oder verringert sich, verändern sich die vorhandenen IP-Adressen für Pods in einem Deployment.

Es wäre daher eine schlechte Idee, wenn Client-Applikationen direkt auf die IP-Adressen von Pods zugreifen würden.

Hier kommen jetzt die Kubernetes Services ins Spiel. Alle Pods eines Deployment können über die IP eines dazwischengeschalteten Service erreicht werden.

Sowohl die IP-Adresse, als auch der DNS-Name und die Portnummer bleibten für einen Service unverändert und können als Schnittstelle von Client-Applikationen genutzt werden, ohne dass die aktuellen IP-Adressen der Pods bekannt sein müssen.



Der Service übernimmt darüber hinaus das Load Balancing, also die gleichmäßige Verteilung der Last an die Pod-Instanzen in einem Deployment.

Das nächste Diagramm stellt diese Zusammenhänge zwischen Pods, Deployment und Service in vereinfachter grafischer Form dar.

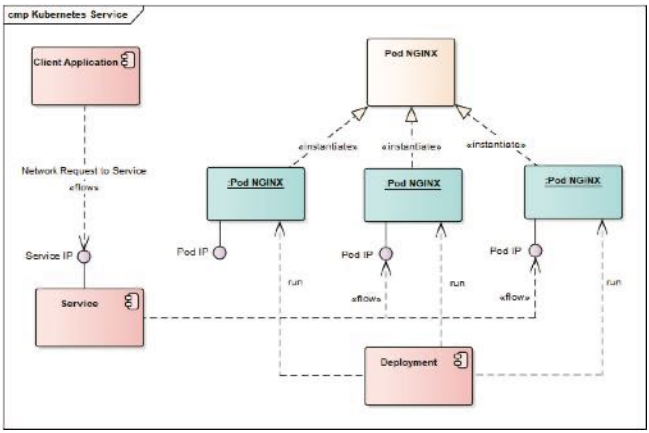



**Abb. 17.3** Service, Deployment und Pods mit ihren Abhängigkeiten

## 17.4 Ein Kubernetes Single Node-Cluster zum Testen und Üben

Nach so viel Theorie bei der Vorstellung der Grundlagen wird es Zeit, Kubernetes auch einmal praktisch kennenzulernen.

Wir führen die ersten Schritte wieder in einer einfachen Single Node-Umgebung durch. Das ist zunächst ausreichend, um den Umgang mit Kubernetes und seinen Kommandos zu üben.

Docker Desktop bietet mittlerweile eine hervorragende Möglichkeit, um ein lokales Entwicklungs-Cluster auf einem Mac- oder Windows-Computer einzurichten. Damit können wir recht schnell und einfach ein Single Node-Cluster für Kubernetes einrichten und darauf dann entwickeln und testen.

Für diejenigen Leser, welche unter Linux arbeiten, gibt es ebenfalls eine Möglichkeit, ein einfaches Single Node-Cluster zu testen und zu üben. Dafür gibt es das Tool Minikube.

Installation und Anwendung von Minikube werden in diesem Buch wieder im Anhang im Kapitel 19.8 behandelt.

### 17.4.1 Kubernetes für Docker Desktop aktivieren

Klicken Sie im Statusbereich der Windows-Taskleiste auf das Docker-Symbol (der kleine Wal), um das Docker Desktop-Menü zu öffnen (Abb. 17.4).

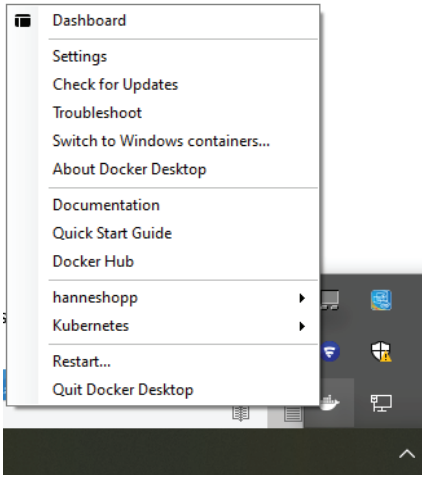



**Abb. 17.4** Menüpunkt SETTINGS im Docker Desktop-Menü

Wählen Sie dort den Menüpunkt SETTINGS aus. Es erscheint das Dialogfenster „SETTINGS“ von Docker Desktop (Abb. 17.5):


**Abb. 17.5** Dialogfenster „SETTINGS“ von Docker Desktop mit KUBERNETES-Eintrag

Falls dort in der linken Auswahlliste der Eintrag KUBERNETES fehlt (siehe Abb. 17.6), dann sind aktuell auf Ihrem Rechner Windows Container für den Docker Daemon aktiviert.

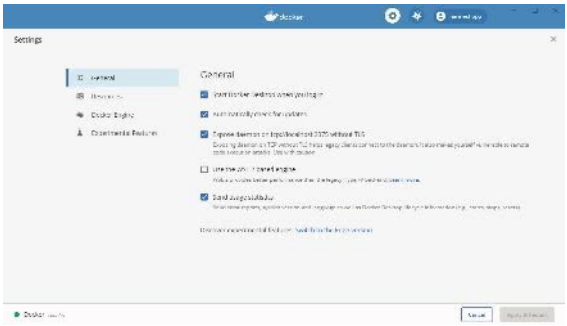



**Abb. 17.6** Dialogfenster „SETTINGS“ von Docker Desktop ohne KUBERNETES-Eintrag

Um Kubernetes zu nutzen, müssen Sie vorher auf Linux Container umschalten. Das geht ebenfalls über das Menü von Docker Desktop (siehe Abb. 17.4). Klicken Sie dort auf den Menüpunkt ‚SWITCH TO LINUX CONTAINERS ...‘ und im Dialogfenster Settings erscheint nach dem Aufruf als letzter Eintrag KUBERNETES (Abb. 17.5).

Klicken Sie auf den Eintrag KUBERNETES, um das zugehörige Register im Settings-Dialog zu aktivieren (Abb. 17.7).

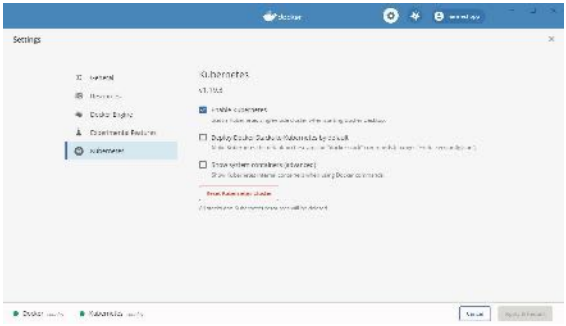



**Abb. 17.7** Register „KUBERNETES“ im Dialogfenster „SETTINGS“ von Docker Desktop

Aktivieren Sie hier das Kontrollkästchen *‚Enable Kubernetes‘*. Durch einen Mausklick auf die Schaltfläche [Apply & Restart] wird die Installation des Kubernetes Clusters gestartet.

Die Installation kann danach eine ganze Weile dauern. Dabei wird am unteren Rand des Dialogfensters rechts neben dem *Docker-Status* (grüner Punkt = *running*) der *Kubernetes-Status* angezeigt (gelber Punkt = *starting*) (Abb. 17.8). Nach erfolgreichem Start wechselt die Anzeige für den Kubernetes-Status von *starting* auf *running*.

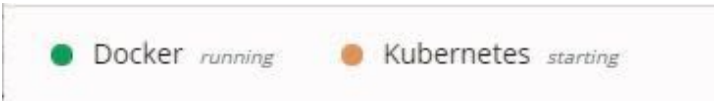



**Abb. 17.8** Statusanzeige im Dialogfenster „Settings“ von Docker Desktop

Damit wären wir soweit und können Kubernetes ausprobieren.

Zum Test starten Sie eine Shell Ihrer Wahl. Dort geben Sie die folgenden Kommandos ein, um die Installation von Kubernetes zu überprüfen (Abb. 17.9). Das verwendete Kommando `kubectl` wird im folgenden Kapitel dann genauer behandelt.

Ausgabe der Versionsinformationen von `kubectl`:

```
1 > kubectl version
```

Cluster-Informationen ausgeben:

```
1 > kubectl cluster-info
```

Node-Informationen ausgeben:

```
1 > kubectl get nodes --all-namespaces
```

Pod-Informationen ausgeben:

```
1 > kubectl get pods --all-namespaces
```

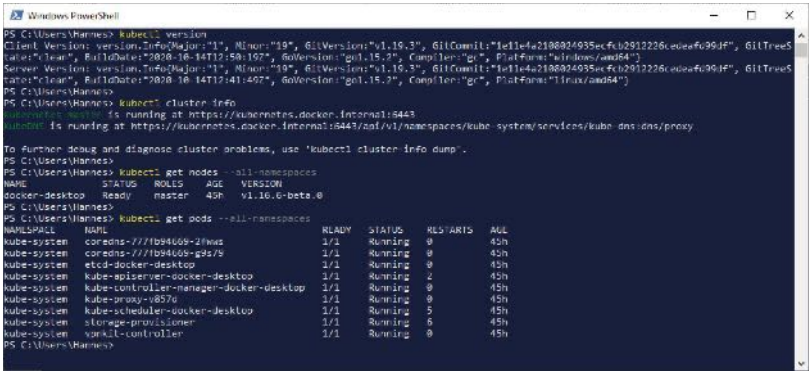



**Abb. 17.9** Die ersten Kubernetes-Befehle in der PowerShell

Wenn bei Ihnen die Befehle mit vergleichbaren Ergebnissen ausgeführt werden, wie im Screenshot zu sehen ist, dann war die Aktivierung des Kubernetes-Clusters auch bei Ihnen erfolgreich.

### 17.4.2 Das Kubernetes-Kommando kubectl

Beim Testen der Installation von Kubernetes haben wir schon eines der wichtigsten Kubernetes-Kommandos kennengelernt – das Kommando `kubectl`. Man kann sich `kubectl` als so etwas wie eine SSH Shell für Kubernetes vorstellen. Es ist die Standard-Kommandozeilen-Anwendung bei der Arbeit mit Kubernetes-Clustern und hat die folgende Syntax:

```
1 kubectl [<command>] [<TYPE>] [<NAME>] [<flags>]
```

Mit dem Parameter `command` spezifiziert man die Operation, die ausgeführt werden soll. Da gibt es zum Beispiel die Kommandos `create`, `get`, `delete` oder `describe`. Zum Testen der Installation haben wir gerade die Kommandos `cluster-info` und `get` verwendet.

Über den Parameter `TYPE` gibt man einen Ressource-Typ an, auf den ein Kommando angewendet werden soll. Ressource-Typen können zum Beispiel `nodes`, `pods`, `deployments` oder `services` sein.

Dem Parameter `NAME` übergibt man den Namen einer Ressource. Wird dieser weggelassen, wird ein Kommando auf alle Ressourcen des angegebenen Typs angewendet.

Optional können noch zusätzliche Flags übergeben werden. Zum Beispiel kann man mit dem Flag `-s` die IP-Adresse und Portnummer eines Kubernetes API Servers angeben. In unseren ersten Beispielkommandos wurde das Flag `--all-namespaces` verwendet. Damit haben wir angegeben, dass Objekte in allen Kubernetes Namespaces gesucht werden sollen. Namespaces bieten bei Kubernetes eine Möglichkeit, um ein Cluster logisch in mehrere virtuelle Cluster aufzuteilen. Damit kann die Verwaltung von Clustern übersichtlicher gestaltet werden.

### 17.4.3 Ein erstes einfaches Deployment

Wir beginnen mit einem ganz einfachen Beispiel. Dabei erzeugen wir ein Deployment eines einzelnen Docker Containers, der auf dem NGINX Image aufbaut. Dieser wird in einer einzelnen Pod-Instanz von Kubernetes bereitgestellt. Zuletzt erzeugen wir einen Service, damit wir eine feste Schnittstelle für den Zugriff auf die Schnittstelle des Pods und damit auf NGINX erhalten.

Das folgende Komponentendiagramm stellt diese Konfiguration grafisch dar (Abb. 17.10):

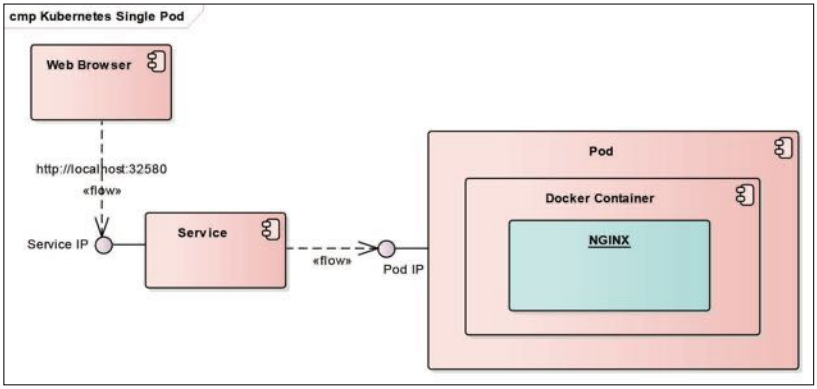



**Abb. 17.10** Komponentendiagramm für ein Deployment mit einem Pod

Wir beginnen mit einem `kubectl`-Kommando, welches einen NGINX Container von Docker in einem Kubernetes Pod bereitstellt:

```
1 > kubectl create deployment nginx-app --image=nginx --port=80
```

Wir verwenden bei diesem Beispiel das `kubectl`-Kommando `create deployment`. Mit dem Parameter --image geben wir hier an, dass ein Docker Container aus dem Image NGINX von Docker Hub erstellt werden soll. Anschließend geben wir `'nginx-app'` als Namen für das neue Kubernetes Deployment an. Mit dem Parameter `--port` legen wir fest, welcher Container Port veröffentlicht wird.

Sehen wir uns das Ergebnis des Kommandos an, indem wir uns eine Liste der Pods im default-Namespace anzeigen lassen:

```
1 > kubectl get pods
```

Anhand der Ausgabe können Sie erkennen, dass die Pod-Namen sich aus dem Namen des Deployments und einer generierten ID zusammensetzen. Bei dieser ID handelt es sich um eine eindeutige ID, die gemäß dem ISO/IEC-Standard für ‚Universally Unique Identifiers' (UUIDs) von Kubernetes gebildet wird.

Mit dem folgenden Kommando prüfen wir, welche Service-Objekte aktuell verfügbar sind:

```
1 > kubectl get services
```

Außer dem Service von Kubernetes selbst gibt es noch keinen weiteren Service im Standard-Namespace.

Wir haben bis jetzt zwar ein Deployment mit einem Pod erstellt, aber es gibt noch keinen Service, der eine einheitliche Schnittstelle für Pod-Zugriffe bereitstellt. Den erzeugen wir mit dem `expose`-Kommando von `kubectl`:

```
1 > kubectl expose deployment nginx-app --name=nginx-svc '
2 --type=NodePort
```

Mit dem Kommando wird der Zugriff auf die interne Pod-Schnittstelle des Deployments `nginx-app` über die unveränderliche Schnittstelle eines neuen Service-Objekts ermöglicht.

Der Parameter `'nginx-app'` bestimmt, dass die Schnittstelle der Pods aus dem oben erstellten Deployment mit diesem Namen veröffentlicht werden soll.

Parameter `--type` bestimmt den Port-Typ. In diesem Beispiel wird `NodePort` als Typ angegeben. Das bedeutet, dass dieser Service-Port für das gesamte Cluster verfügbar sein soll.

Mit dem Parameter `--name` bekommt der Service den Namen `'nginx-svc'`.

Fragen wir jetzt noch einmal die Liste der Services ab:

```
1  > kubectl get services
```

Zusätzlich zum Service von Kubernetes erscheint jetzt der neue Service 'nginx-svc' mit den zugehörigen Informationen in der ausgegebenen Liste.

Hier der Screenshot mit den gerade vorgestellten Kommandos in Aktion (Abb. 17.11):

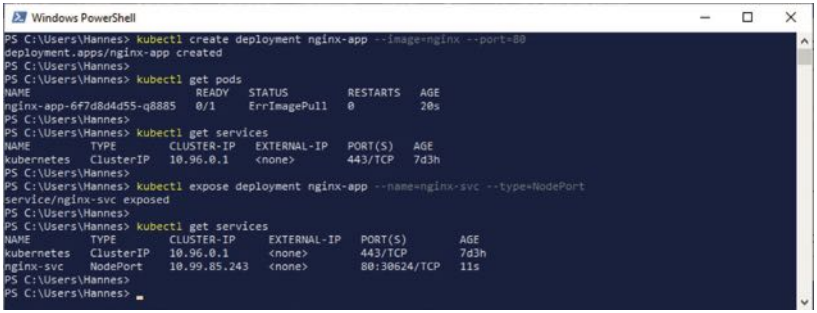



**Abb. 17.11** Ein einfaches Deployment mit dem Kommando `kubectl create`

Um detaillierte Informationen über eine Ressource von Kubernetes zu erhalten, wird über `kubectl` das Kommando `describe` bereitgestellt. Mit dem Parameter `svc` (oder auch `service`) geben wir an, dass wir Informationen von einem Service wollen. Als letzten Parameter setzten wir den Namen des Service ein. Ohne dessen Angabe würden die Informationen von allen Services im Namespace ‚default' angezeigt (Abb. 17.12).

```
1  > kubectl describe svc nginx-svc
```

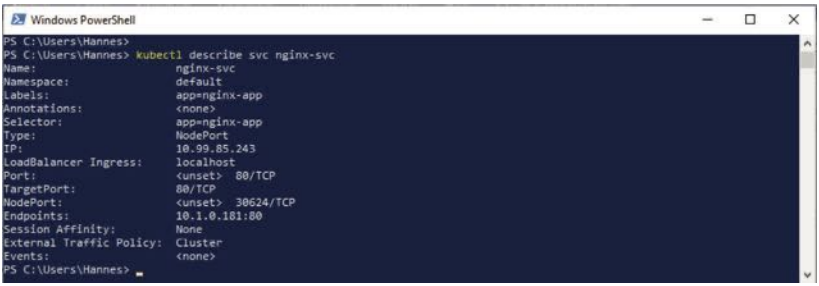



**Abb. 17.12** Detaillierte Informationen über einen Kubernetes Service mit dem Kommando `kubectl describe`

Natürlich wollen wir zuletzt auch noch auf die Anwendung zugreifen, um zu sehen, ob unser Deployment auch wirklich funktioniert. Dazu müssen wir eigentlich nur die Portnummer des Service herausfinden. Wir haben diese Information sowohl über das Kommando kubectl get services als auch mit dem zuletzt ausgeführten Kommando kubectl describe erhalten. Dort wird in den Screenshots aus dem Beispiel für den Node Port die Portnummer 30624 angezeigt. Wenn Sie das Beispiel ausgeführt haben, dann wird bei Ihnen dort wahrscheinlich eine andere Portnummer zugewiesen worden sein.

Finden Sie also die Portnummer für Ihr Deployment heraus.

Mit dem `curl`-Kommando können Sie jetzt aus einer Shell heraus mit dieser Information auf die Schnittstelle des NGINX Containers zugreifen (Abb. 17.13):

```
1  > curl http://localhost: 30624
```

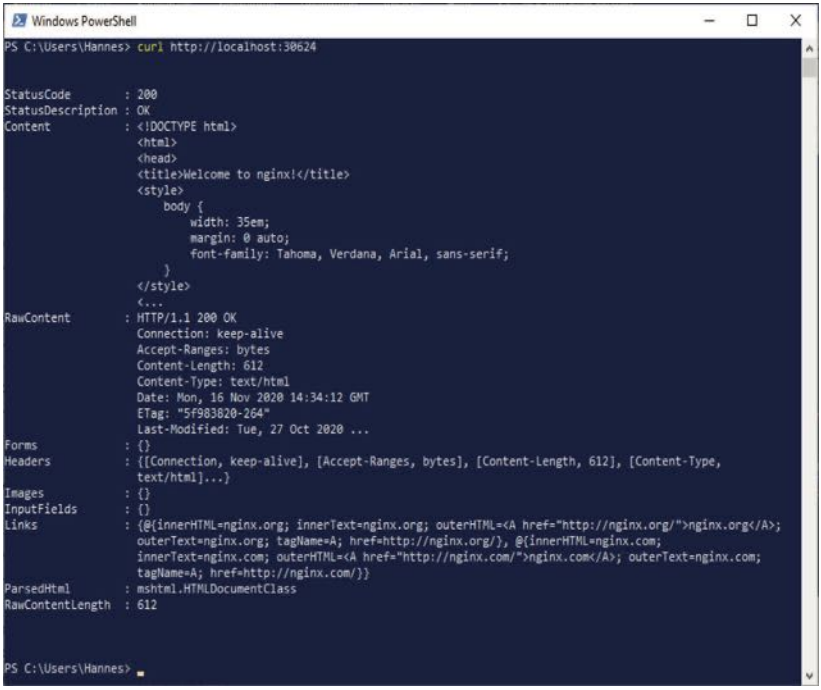



**Abb. 17.13** Zugriff über einen Kubernetes Service auf einen Container mit `curl`

Wenn Sie die gleiche URL in der Adresszeile eines Browsers eingeben, dann erscheint wieder die bereits bekannte ‚Welcome to nginx'-Webseite aus dem NGINX Container (Abb. 17.14):

```
1  http://localhost:<port_nummer>
```

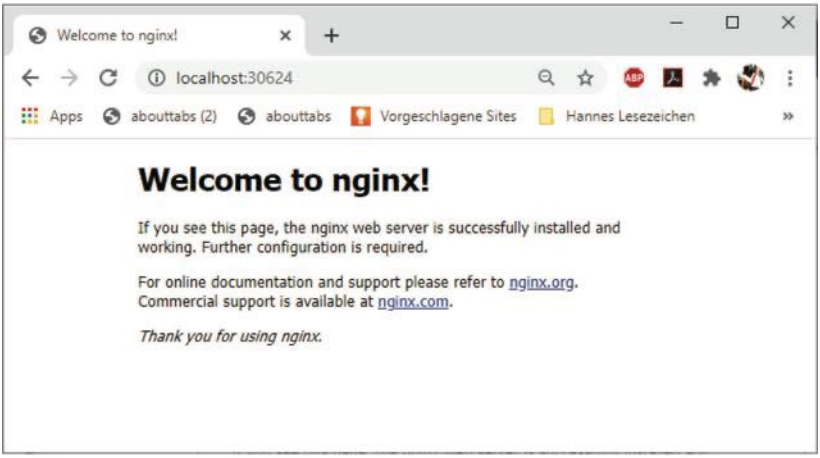



**Abb. 17.14** Browser-Zugriff auf den NGINX Container über einen Kubernetes Service

Wie immer räumen wir auf, indem wir die erstellten Objekte aus dem Beispiel wieder entfernen. Dabei lernen wir auch noch die dafür nötigen Kommandos von `kubectl` in der praktischen Anwendung kennen.

Lassen wir uns erst noch eine Liste mit allen Deployments ausgeben (Abb. 17.15):

```
1  > kubectl get deployment
```


**Abb. 17.15** Anzeige der aktuellen Deployments

Wir sehen hier unser Deployment mit dem Namen '`nginx-app`'.

Zum Löschen geben wir das `kubectl`-Kommando `delete` an, danach folgt als Typ `deployment` und als letztes Argument übergeben wir den Namen des Deployments ‚`nginx-app`'. Anschließend sehen wir noch einmal nach, ob das Deployment wirklich entfernt wurde:

```
> kubectl delete deployment nginx-app
> kubectl get deployment
```

Sehen wir jetzt nach, was mit unserem Service passiert ist, und geben dafür noch einmal die Liste der Services aus:

```
> kubectl get services
```

Wie Sie sehen können, haben wir zwar das Deployment gelöscht, aber das Service-Objekt existiert immer noch und man muss es eigens löschen.

Das erledigen wir wieder mit dem `delete`-Kommando von `kubectl`, geben jetzt aber als Objekttype `service` an und den Namen `'nginx-svc'`, welchen wir unserem Service gegeben haben. Anschließend prüfen wir das Ergebnis des Kommandos:

```
> kubectl delete svc nginx-svc
> kubectl get services
```

Sicher wollen Sie jetzt noch wissen, ob auch Pods separat gelöscht werden müssen. Sehen wir also nach, ob unser Pod noch existiert:

```
> kubectl get pods
```

Als Ergebnis kommt die Meldung "`No resources found in default namespace`". Damit wird klar, dass unser Pod zusammen mit seinem Deployment entfernt wurde.

Die Kommandos aus dem Beispiel, ausgeführt in einer PowerShell, hier noch einmal als Screenshot (Abb. 17.16):

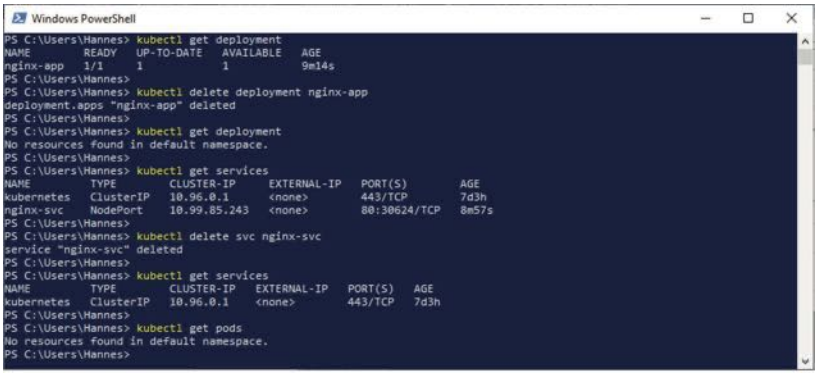



**Abb. 17.16** Kommandos zum Entfernen eines Deployments und seines Service

### 17.4.4 Die Deployment Manifest YAML-Datei

Ein Kubernetes Deployment mit dem `run`-Kommando zu erstellen ist nicht die empfohlene Methode, um Applikationen mit Kubernetes bereitzustellen. Sie sollte eigentlich nur für Tests während der Entwicklung einer Anwendung genutzt werden.

Die große Stärke von Kubernetes ist, dass es auf dem sogenannten *deklarativen Model* aufbaut. Was bedeutet das?

Bei einem *deklarativen Modell* gibt man an, was man erreichen möchte, also wie das gewünschte Ergebnis aussehen soll. Das Gegenstück wird als *imperatives Modell* bezeichnet. Dort gibt man detaillierte Anweisungen für jeden Schritt an, der ausgeführt werden soll.

Beispiele für das *imperative Modell* sind Ihnen sicher bereits geläufig. Im Grunde genommen gehören die Anweisungen des Quellcodes aller Programmier- oder Skriptsprachen zu dieser Kategorie.

Wir haben aber in diesem Buch schon ein Beispiel für den deklarativen Ansatz kennengelernt. Docker Stack basiert nämlich ebenfalls auf dem deklarativen Modell. In einer YAML-Datei trägt man ein, was man haben

möchte, zum Beispiel wie viele Replikas von den jeweiligen Containern vorhanden sein sollen. Docker Stack kümmert sich dann um alle dafür notwendigen Aktivitäten. Dabei wird zum Beispiel auch dafür gesorgt, dass im Fehlerfall ein neues Replikat eines Containers instanziiert wird, um eine defekte Instanz zu ersetzen.

Bei Kubernetes werden die Information für ein Deployment in einer Datei verwaltet, die man auch Manifest nennt. Wie schon bei Docker Compose und Docker Stack handelt es sich hier wieder um eine Datei im YAML-Format. Sie folgt allerdings einer anderen Struktur.

Auf oberster Ebene gibt es dabei die folgenden vier Sektionen:

| | |
|---|---|
| `kind:`<br><br>Beispiel<br><br>`kind: Pod` | Mit diesem Element geben wir an, um welche Objektart es sich hier handelt, d.h. was für ein Objekt wollen wir mit diesem Eintrag spezifizieren. Folgende Einträge sind hier möglich: Deployment, Pod, Service, Secret, Jobs, Replication Controller und Replicaset. Jede Art der Manifest-Datei dient einem speziellen Zweck. |
| `apiVersion:`<br><br>Beispiel<br><br>`apiVersion: apps\v1beta1` | Dieser Eintrag ist abhängig von dem aktuellen Eintrag in der Sektion `'kind'`. Er bestimmt zwei Dinge. Zum einen wird die API-Gruppe, zum anderen die API-Version festgelegt.<br><br>Für jeden Manifest-Typ, der mit `'kind'` angegeben werden kann, existieren eigene API-Gruppen und eigene API-Versionen.<br><br>Im Beispiel links steht als Gruppe `'apps'` und als Version `'v1'`. Das ist eine Angabe, die bei einem Manifest-Objekt vom Typ `'Deployment'` möglich wäre. |

| | |
|---|---|
| `metadata:`<br><br>Beispiel<br><br>`metadata:`<br><br>`name:`<br>`nginx-app` | Unter dieser Sektion können Namen und Labels definiert werden. Diese sind hilfreich, um Objekte in einem Cluster leichter zu identifizieren. Es ist auch möglich, hier einfache Verbindungen zwischen den Objekten eines Clusters herzustellen.<br><br>Das Beispiel definiert den Namen `'nginx-app'` als Objektnamen. |
| `spec:` | Unterhalb dieser Sektion beginnen die Spezifikationen für das Kubernetes-Objekt. Hier kann man zum Beispiel die Anzahl der Replikate festlegen oder welche Container integriert werden. Die konkreten Angaben, die hier gemacht werden können, hängen letztendlich vom Objekttyp ab, der in der Sektion Kind festgelegt wurde. |

### 17.4.5 Ein einfaches Deployment deklarativ erstellen

Unser erstes Deployment haben wir mit den `kubectl`-Kommandos `run` und den Service mit dem Kommando `expose` erstellt. Benötigte Angaben müssen hier bei der Ausführung der Kommandos als Parameter übergeben werden.

Jetzt tragen wir alle Informationen in eine YAML-Datei ein, die Manifest-Datei.

Der Name dieser Datei wird dann dem `kubectl`-Kommando `create` als Parameter übergeben.

#### *17.4.5.1 Die YAML-Datei des Deployments*

Wir beginnen mit der YAML-Datei für das Deployment. Dieses soll im ersten Schritt die gleichen Eigenschaften besitzen wie das Deployment ‚nginx-app' aus dem ersten Beispiel.

Die Datei soll den Dateinamen `'nginx-app.yaml'` erhalten.

Die Dateien für das Beispiel legen wir in einem neuen Arbeitsverzeichnis an. Mein Vorschlag dafür wäre, dass Sie ein eigenes Verzeichnis für alle Kubernetes-Übungen unter Ihrem Benutzerverzeichnis anlegen.

```
<USER_DIR>\Kubernetes
```

Darunter legen wir dann die Verzeichnisse der verschiedenen Übungsbeispiele an. Das Verzeichnis für das erste Beispiel erhält hier den Namen `'nginx-app'`.

```
<USER_DIR>\Kubernetes\nginx-app
```

Erzeugen sie dort die Manifest-Datei `'nginx-app.yaml'` mit folgendem Inhalt:

```
Datei 'nginx-app.yaml'

apiVersion: apps/v1

kind: Deployment

metadata:
  name: nginx-app
  labels:
    app: nginx

spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

In dieser YAML-Datei wird im Eintrag `apiVersion` die API `apps` in der Version `v1` angegeben. Das ist die am häufigsten genutzte API-Gruppe von Kubernetes. Sie bietet die wichtigsten Funktionalitäten, um Objekte wie Deployments mit Kubernetes zu definieren.

Mit dem Eintrag `kind: Deployment` informieren wir Kubernetes, dass diese Datei zum Erstellen eines Deployment-Objekts dient.

In der Sektion `metadata` geben wir dem Deployment den Namen `'nginx-app'` und definieren noch unter `labels` ein Label mit dem Key `'app'` und dem Value `'nginx'` (`'app: nginx'`) für das Deployment.

In der Sektion `spec:` folgt die Spezifikation des Deployments.

Mit dem Eintrag `'replicas: 1'` wird angegeben, dass es nur ein Replikat des Pod-Objekts geben soll.

Unter dem Abschnitt `selector` befinden sich die Informationen darüber, welche Pods durch dieses Deployment verwaltet werden. Durch die Angabe von `matchLabels` wird dann festgelegt, dass es die Pods-Deklarationen sind, denen das Label `'app: nginx'` zugeordnet ist.

Die `template`-Sektion dient der Konfiguration der Pods. Darunter gibt es wieder einen Abschnitt `metadata`, unter dem als Label für die Pods `'app: nginx'` definiert wird. Dieses Label wird vom Deployment-Selektor ausgewertet.

Unterhalb der `template`-Sektion befindet sich noch ein weiterer Abschnitt `spec`. Dort liegen die Spezifikation der Container für diesen Pod. Diese befinden sich unter dem Element `containers` und legen die Namen der Container fest – mit den zu verwendenden Images und deren Version. Im Beispiel wird nur ein Container mit dem Namen `nginx` angegeben. Dieser basiert auf dem Image `nginx` in der Version 1.19.1. Als Letztes, unter dem Element `ports` wird Port 80 als Container-Port angegeben.

### *17.4.5.2 Ein Deployment mit create erstellen*

Mit dem Ausfüllen der Manifest-Datei ist dann schon die Hauptarbeit bei der Entwicklung eines Deployments erledigt. Um es zu erzeugen, muss die YAML-Datei dem `kubectl`-Kommando `create` als Parameter übergeben werden. Die Syntax dafür ist recht übersichtlich:

```
1  kubectl create -f <DATEINAME>
```

Mit dem Parameter `-f` oder `-filename` übergibt man den Pfad zur Manifest-Datei. Hier kann aber auch eine URL angegeben werden.

Wir starten eine Shell und wechseln dort in das Verzeichnis, in dem sich die Datei `'nginxapp.yaml'` befindet. Dann geben wir das folgende Kommando ein:

```
1  > kubectl create -f .\nginx-app.yaml
```

Wenn in der YAML-Datei keine Fehler sind und auch sonst keine Probleme aufgetreten sind, dann wird das Kommando mit der folgenden Meldung quittiert:

```
1  "deployment .apps/nginx-app created"
```

Mit dem Kommando `get` lassen wir uns wieder die aktuell verfügbaren Deployments als Liste ausgeben:

```
1  > kubectl get deployment
```

Detaillierte Informationen zum Deployment erhalten wir durch dieses Kommando:

```
1  > kubectl describe deployment nginx-app
```

Im Anschluss sehen Sie wieder den Screenshot einer PowerShell mit den beschriebenen Kommandos (Abb. 17.17):

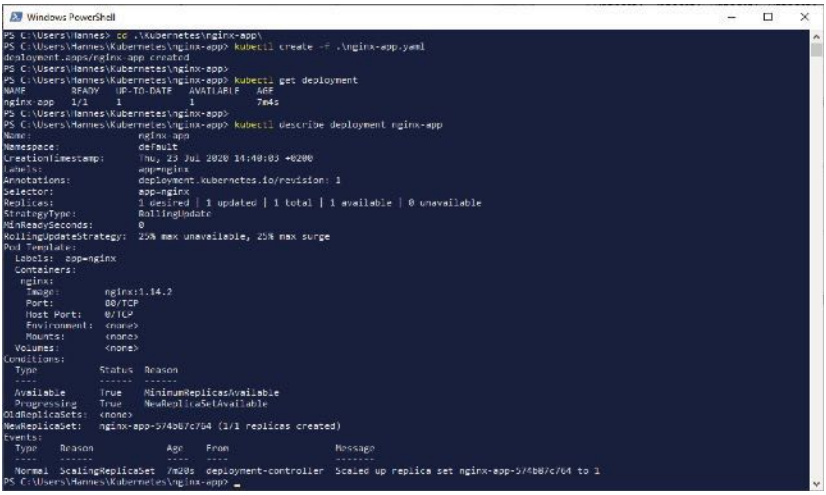



**Abb. 17.17** Erzeugen eines Deployments aus einer YAML-Manifest-Datei

### 17.4.6 Einen Service mit YAML erstellen

Damit haben wir unser Deployment und es fehlt nur noch ein Service Objekt um die einheitliche und unveränderliche Schnittstelle zum Deployment bereitzustellen.

Auch die Eigenschaften von Services können über Manifest-Dateien festgelegt werden. Das gilt übrigens für alle Objektarten von Kubernetes.

Der Name der Manifest-Datei wird dann, genauso wie schon beim Deployment, dem `kubectl` Kommando `create` als Parameter übergeben.

#### *17.4.6.1 Die YAML-Datei*

Die Manifest-Datei des Service erhält den Dateinamen `'nginx-svc.yaml'`.

Sie wird ebenfalls im Verzeichnis unseres NGINX-Beispiels abgespeichert:

```
<USER_DIR>\Kubernetes\nginx-app
```

Tragen Sie in diese Datei die folgenden Angaben ein:

```
Datei 'nginx-svc.yaml'

apiVersion: v1

kind: Service

metadata:
  name: nginx-svc
  labels:
    app: nginx

spec:
  type: NodePort
  ports:
  - nodePort: 32580
    port: 80
    targetPort: 80
  selector:
    app: nginx
```

Der erste Eintrag gibt als `apiVersion` `v1` an. Das ist die erste stabile API-Version von Kubernetes. Diese Version enthält die Schnittstellen für zahlreiche Basis-Objekte von Kubernetes, so auch die von Services.

Der Eintrag `'kind: Service'` gibt an, dass man mit dieser Datei ein Service-Objekt erstellen möchte.

In der Sektion `metadata` geben wir dem Service den Namen `'nginx-svc'` und definieren noch das Label `'app: nginx'` für den Service.

In der Sektion `spec:` folgt hier die Spezifikation für einen Service.

Der Service-Typ NodePort (`type: NodePort`) bewirkt, dass der Service über einen statischen Port von jedem Node bereitgestellt wird.

Dadurch wird der Zugriff auf diesen Service auch von außerhalb des aktuellen Clusters möglich.

Unter der Sektion `ports` wird eine Liste mit den Port-Typen für einen Service angelegt. Dabei sind drei Angaben möglich. Die Portnummer, welche mit dem Port-Typ `'port'` angegeben wird, stellt der Kubernetes Service innerhalb des Clusters bereit. Andere Pods innerhalb desselben Clusters können mit dem Service über diesen Port kommunizieren.

Der Typ `nodePort` definiert die Portnummer für Zugriffe auf den Service von außerhalb des Clusters. Der zulässige Wertebereich für diese Portnummer geht von 30000 bis 32767.

Mit dem Port-Typ `targetPort` wird die Portnummer des Containers angegeben.

#### *17.4.6.2 Einen Kubernetes Service mit create erstellen*

Starten Sie wieder eine Shell und wechseln in das Verzeichnis für unser NGINX-Beispiel. Dort sollte sich auch die Datei `,nginxsvc.yaml‘` befinden. Den Service erzeugen wir ebenfalls mit dem `create`-Kommando von `kubectl`:

```
1 > kubectl create -f .\nginx-svc.yaml
```

Wenn alles in Ordnung ist, dann gibt `kubectl` die folgende Meldung aus:

```
1 "service/nginx-svc created"
```

Wir lassen wir uns die aktuellen Services auflisten:

```
1 > kubectl get svc
```

Mit dem `curl`-Kommando prüfen wieder den Zugriff auf den NGINX Container über den Service. Als Portnummer muss jetzt diejenige an-

gegeben werden, die in der YAML-Datei unter dem Eintrag nodePort definiert wurde:

```
1  > curl http://localhost:32580
```

Auch hier noch ein Screenshot mit den vorgestellten Aktionen (Abb. 17.18):

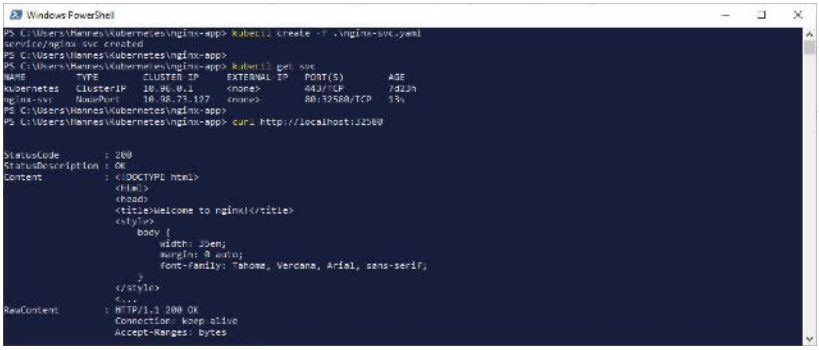



**Abb. 17.18** Erzeugen eines Service aus einer YAML-Manifest-Datei

Wenn das `curl`-Kommando die im Screenshot gezeigte Ausgabe liefert, dann sollte der Zugriff auf die NGINX-Webseite auch über einen Webbrowser funktionieren.

### 17.4.7 Ein laufendes Deployment modifizieren

Um eine gestartete Applikation bei laufendem Betrieb zu modifizieren, haben wir bei Docker Swarm das Kommando `docker service update` kennengelernt. Wird Docker Stack zur Verwaltung eines Deployments eingesetzt, dann kann mit dem Kommando `docker stack deploy` ein neuer Stack bereitgestellt werden, oder ein existierender Stack wird damit aktualisiert.

Bei Kubernetes wird eine vergleichbare Funktion mit dem `kubectl`-Kommando `apply` zur Verfügung gestellt. Damit können Änderungen, die zuvor in einer zugehörigen Manifest-Datei vorgenommen wurden,

an aktive Kubernetes-Objekte wie Deployments oder Services übergeben und dort in Kraft gesetzt werden.

Für dieses Kommando gilt die folgende Syntax:

```
kubectl apply -f <FILENAME> [FLAG ...]
```

Zusammen mit dem Parameter `-f` wird bei diesem Kommando die Manifest-Datei mit den Änderungen angegeben. Als Flag lernen Sie in diesem Kapitel noch die Angabe `--record` in Verbindung mit Rolling Updates kennen

#### *17.4.7.1 Die Anzahl der Pod-Replikate ändern*

In Kapitel 17.4.5 haben wir ein Deployment mit dem Namen `'nginx-app'` erstellt. Dort wurde für die Anzahl der Pod-Replikate der Wert 1 angegeben.

Falls das Deployment- und das Serviceobjekt aus diesem Beispiel in der Zwischenzeit gelöscht worden sind, dann sollten diese für das folgende Beispiel wieder aktiviert werden, denn es soll ja hier demonstriert werden, wie eine laufende Anwendung modifiziert werden kann.

Mit unserem ersten Beispiel erhöhen wir für das Deployment `'nginx-app'` die Anzahl der Replikate auf 6. Dazu ändern wir als Erstes in der Manifest-Datei `'nginx-app.yaml'` unter der Sektion `spec` den Eintrag `replicas`. Wir setzen den Wert auf 6 und speichern die Änderung ab (!!).

```
spec:
  replicas: 6
```

Wir öffnen eine Shell und sorgen dafür, dass wir uns im Verzeichnis des NGINX-Beispiels befinden, wo auch die Datei `'nginxapp.yaml'` liegt. Die Aktualisierung des Deployments wird jetzt mit dem `apply`-Kommando von `kubectl` gestartet:

```
> kubectl apply -f .\nginx-app.yaml
```

Die Aktualisierung des Deployments kann eine ganze Weile dauern, weil die neuen Pod-Replikate der Reihe nach gestartet werden und der alte Pod gelöscht wird.

Wir können in der Zwischenzeit den Rollout-Status an der Konsole durch Eingabe eines `kubectl`-Kommandos mitverfolgen:

```
1  > kubectl rollout status deployment nginx-app
```

Zuletzt führen wir das `kubectl`-Kommando `get pod` aus, um das Ergebnis des `apply`-Kommandos zu sehen:

```
1  > kubectl get pod
```

Damit haben wir für das Deployment die Anzahl der Pod-Instanzen auf 6 erhöht. Das Ganze noch einmal in einem Beispiel-Screenshot (Abb. 17.19):

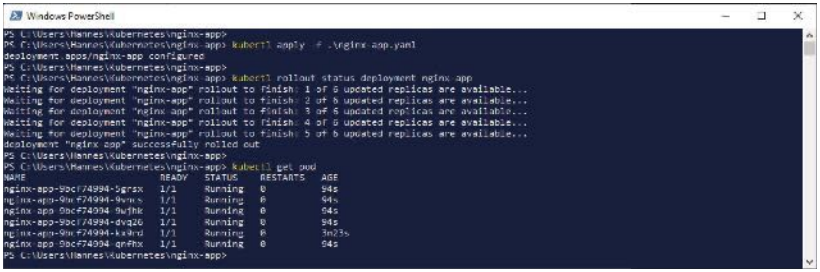



**Abb. 17.19** Erhöhung von Pod-Replikaten durch das Kommando `kubectl apply`



### *17.4.7.2 Anwendung mit Rolling Updates aktualisieren*

Mit dem `kubectl apply`–Kommando erlaubt es Kubernetes, sogenannte „Rolling Updates“ durchzuführen. Was hat das zu bedeuten?

Rolling Updates ermöglichen die Aktualisierung von Deployments, ohne dass Ausfallzeiten für den Anwender entstehen.

Dabei werden die Pod-Instanzen inkrementell. Das heißt, dass eine Instanz nach der anderen durch eine neue Instanz mit der neuen Version ersetzt wird. Während dieser Zeit wird der Datenverkehr nur auf die verfügbaren Pods verteilt.

Durch Rolling Updates ist es auch möglich, Rollbacks auszuführen, also auf eine Version von früheren Deployments zurückzuspringen.

Soll ein späteres Rollback ermöglicht werden, muss man das `apply`-Kommando noch um den Parameter `--record` ergänzen. Damit wird von Kubernetes der Verlauf der Revisionen für das Deployment gespeichert.

```
> kubectl apply (-f <FILENAME> | -k <DIRECTORY>) --record
```

Um ein „Rolling Update“ zu demonstrieren, ändern wir bei unserem Deployment ‚nginx-app‘ die Version von NGINX auf 1.18.0.

Das geschieht ganz einfach, indem man die Versionsangabe für das zu verwendende NGINX Image in der YAML-Datei ändert:

```
...
      containers:
      - name: nginx
        image: nginx:1.18.0
...
```

Diese Änderung würde schon genügen. Es würde so ein Update des Deployments mit Standard-Einstellungen ausgeführt werden. Wenn wir aber die Art und Weise, mit der das Update ausgeführt werden soll, genauer definieren wollen, dann können wir in der Manifest-Datei die Sektion `spec` des Deployments um weitere Angaben für eine Update-Strategie erweitern.

So sieht die Manifest-Datei mit den geänderten Angaben aus:

```
Datei 'nginx-app.yaml'

apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: nginx-app
  labels:
    app: nginx
spec:
  replicas: 6
  selector:
    matchLabels:
      app: nginx

  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1

  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.18.0
        ports:
        - containerPort: 80
```

Der Eintrag `'minReadySeconds'` bestimmt, wie viele Sekunden zwischen den Aktualisierungen der einzelnen Pods gewartet werden soll.

Danach wird hier angegeben, dass als Update-Strategie „RollingUpdate“ zum Einsatz kommen soll.



Mit dem Eintrag `'maxUnavailable: 1'` erreichen wir, dass während eines Updates die Anzahl der aktiv laufenden Pod-Replikas niemals kleiner als der Soll-Status (die gewünschte Anzahl von Replikas) *minus 1* wird.

Der Eintrag `'maxSurge: 1'` bewirkt, dass während des Updates die Anzahl der Replikas niemals größer als die gewünschte Anzahl *plus 1* wird.

In unserer Beispieldatei hat die Anzahl der Replikas den Sollwert 6. Mit den angegebenen Werten für `maxUnavailable: 1` und `maxSurge:1` haben wir sichergestellt, dass während eines Updates immer 5 bis 7 Pod-Instanzen verfügbar sind.

Durch den erneuten Aufruf des apply-Kommandos werden diese Änderungen als Rolling Update auf das laufende Deployment angewendet:

```
1  > kubectl apply -f .\nginx-app.yaml --record
```

Wir verfolgen den Rollout-Status:

```
1  > kubectl rollout status deployment nginx-app
```

Dann sehen wir uns wieder die Deployment-Informationen an:

```
1  > kubectl get deploy nginx-app
```

Zuletzt kommt die Ausgabe der Liste mit Pod-Informationen:

```
1  > kubectl get pod
```

Hier der Screenshot mit dem Beispiel des beschriebenen „Rolling Update“ (Abb. 17.20).

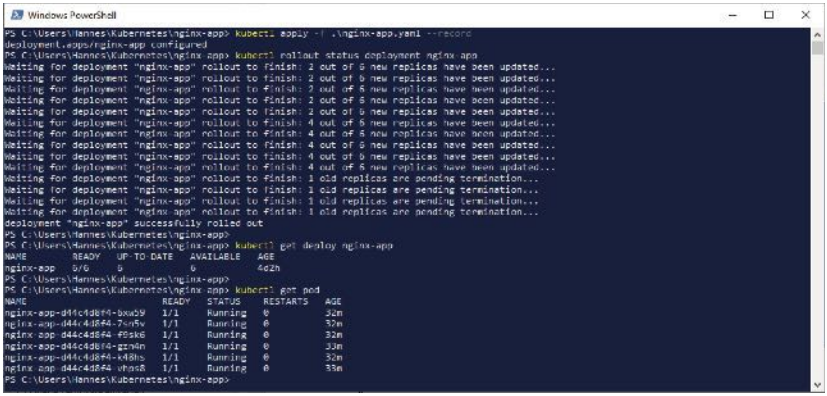



**Abb. 17.20** „Rolling Update“ durch das Kommando `kubectl apply`

### *17.4.7.3 Übungsaufgabe: Deployment ändern*

Zur Übung führen Sie die folgende Aufgabe durch:

Bearbeiten Sie die Manifest-Datei für das Deployment von `'nginx-app'` so, dass folgende Änderungen wirksam werden:

- Es sollen nur noch 3 Pod-Replikate aktiv sein.
- Es soll das NGINX Image mit der Version 1.19.1 angewendet werden.

Führen Sie anschließend ein Rolling Update mit der geänderten Manifest-Datei aus.

Überprüfen Sie zuletzt das Ergebnis.

***Lösung:***

Die Änderungen der Manifest-Datei `'nginx-app.yaml'`

```
Datei 'nginx-app.yaml'

...

spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx

...
    spec:
      containers:
      - name: nginx
        image: nginx:1.19.1
...
```

Start des Updates mit den Änderungen:

```
> kubectl apply -f .\nginx-app.yaml --record
```

Überprüfung des Rollout-Status:

```
> kubectl rollout status deployment nginx-app
```

Überprüfung der Pod-Replikas:

```
> kubectl get pod
```

Die Version von NGINX zusammen mit den Informationen zum Deployment anzeigen:

```
> kubectl describe deployment nginx-app
```

Hier der Screenshot zur Lösung (Abb. 17.21):

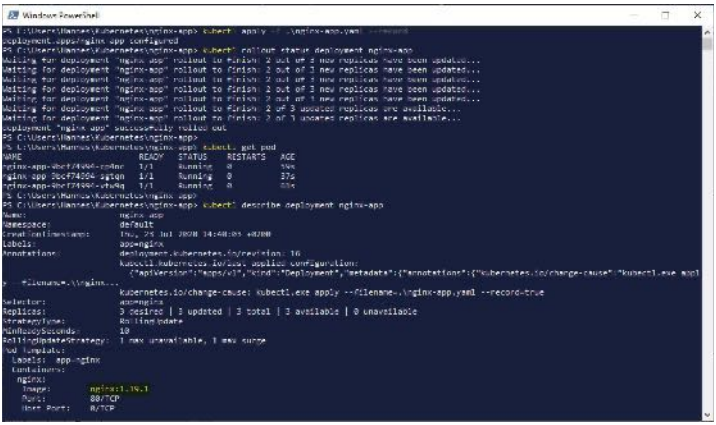


**Abb. 17.21** „Rolling Update"-Lösung zur Übungsaufgabe

### *17.4.7.4 Hier geht es weiter*

Es kann immer wieder vorkommen, dass nach einem Update irgendwelche Probleme auftreten und man wieder zum letzten funktionierenden Stand der Applikation zurückkehren möchte.

Für Docker Services haben wir zu diesem Zweck das Kommando `docker service rollback` kennengelernt.

Bei Kubernetes wird diese Aufgabe mit dem `kubectl`-Kommando `rollout undo` erledigt:

```
1  kubectl rollout undo (TYPE NAME | TYPE/NAME) [flags]
```

Das Kommando benötigt als Parameter den Objekttyp und den Objektnamen. Dabei können diese Angaben durch ein Leerzeichen oder durch einen Schrägstrich getrennt werden. Bei unserem NGINX-Beispiel könnte man demnach `'deployment nginx-app'` oder alternativ `'deployment/nginxapp'` schreiben.

Als Flag ist hier unter anderem die Angabe von `'--to-revision=<REV>'` möglich. So kann man angeben, auf welche Version aus der Historie zurückgesprungen werden soll. Wird dort der Wert 0 zugewiesen,

dann folgt ein Rücksprung auf die letzte Version. Wird dieses Flag ganz weggelassen, dann wir der Wert 0 als Standardwert eingesetzt.

Wenn wir einen Überblick über die Rollout-Historie haben wollen, dann bietet Kubernetes dafür das Kommando `rollout history` an:

```
1 kubectl rollout history (TYPE NAME | TYPE/NAME)
```

Für die Angabe von Objekttyp und Objektname gelten die gleichen Regeln wie beim Kommando `rollout undo`.

Lassen wir uns also erst einmal die Rollout-Historie mit dem folgenden Kommando anzeigen:

```
1 > kubectl rollout history deployment nginx-app
```

Jetzt springen wir zur letzten Version zurück:

```
1 > kubectl rollout undo deployment nginx-app --to-revision=0
```

Den Rollback-Status können wir wie gewohnt verfolgen:

```
1 > kubectl rollout status deployment nginx-app
```

Um das Ergebnis zu prüfen, lassen wir uns wieder für das Deployment die aktuell laufende Version von NGINX zusammen mit weiteren Informationen anzeigen:

```
1 > kubectl describe deployment nginx-app
```

Auch für die Rollback-Funktion folgt hier noch einen Beispiel-Screenshot (Abb. 17.22).

Bei diesem Screenshot kann man erkennen, dass ein Rollback auf die Image-Version eines früheren Deployments durchgeführt wird. Die Anzahl der aktuell laufenden Replikate wird dadurch aber nicht beeinflusst.

Nebenbei bemerkt: Bei einem Rollback wird eine Imperative Operation ausgeführt. Der Inhalt der Manifest-Datei wird dadurch nicht verändert und ihr Inhalt entspricht daher auch nicht dem aktuellen Status des laufenden Deployments.

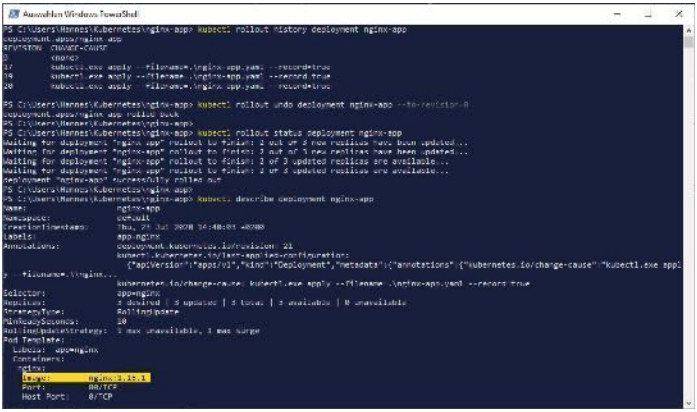



**Abb. 17.22** „Rolling Update“ Rollback zu einem früheren Deployment

Damit sind wir am Ende der Einführung in die Arbeit von Kubernetes auf der Basis von lokalen Single Node-Clustern. Aber diese Cluster-Art eignet sich nur zum Lernen und Üben oder während der Entwicklung von Applikationen zum Testen. Zum Bereitstellen einer Anwendung benötigen wir letztendlich Multi Node-Cluster. Nur dort bringen die Stärken von Kubernetes tatsächlich die gewünschten Vorteile.



## 17.5 Multi Node-Cluster mit Kubernetes

In diesem Kapitel erfahren Sie, wie man bei Kubernetes für ein Multi Node-Cluster den Manager Node erstellt, wie man dem Cluster Worker Nodes zufügt und wie auf so einem Cluster Applikationen bereitgestellt werden.

Wir geben Ihnen dazu eine Übersicht über die gängigsten Cloud-Plattformen, auf denen „Hostet Kubernetes Services“ angeboten werden.

Eine davon, nämlich die „Google Cloud Platform“, ziehen wir dann heran, um das Deployment einer einfachen Web-Anwendung in einer Cloud-Umgebung mit einem praktischen Beispiel vorzustellen.

Falls Sie aber den Umgang mit Kubernetes Multi Node-Clustern ohne viel Aufwand üben wollen und ohne, dass Sie sich bei einem Cloud Service Provider registrieren müssen, dann empfehlen wir Ihnen ein Online Tool, welches im Grunde genommen genauso gehandhabt wird, wie das Tool „Play with Docker“. Dieses Tool heißt, welch eine Überraschung, „Play with Kubernetes“.

Auch dafür haben wir im Anhang das Kapitel 19.7 angelegt, das Ihnen eine Anleitung zur Arbeit mit diesem Tool anbietet.

### 17.5.1 Hosted Kubernetes

Die meisten Cloud-Plattformen bieten ihren eigenen ‚Hostet Kubernetes‘ Service an. Dabei wird in der Regel der Manager Node mit dem Control Layer durch die Cloud-Plattform selbst verwaltet. Das bedeutet für den Anwender zwar weniger Kontrolle über ein Deployment, andererseits reduziert sich aber der Verwaltungsaufwand sehr deutlich. Oft kann mithilfe dieser Dienste das Deployment eines Kubernetes-Clusters mit wenigen Mausklicks bewerkstelligt werden.

Hier eine Auswahl von Cloud-Plattformen für „Hosted Kubernetes Services“:

- AWS: Elastic Kubernetes Service (EKS)
- Azure: Azure Kubernetes Service (AKS)
- DigitalOcean: DigitalOcean Kubernetes
- Rackspace KAAS
- IBM Cloud: IBM Cloud Kubernetes Service
- Google Cloud Platform: Google Kubernetes Engine (GKE)

Da wir im Rahmen dieses Buches nicht alle Plattformen vorstellen können beschränken wir uns auf die Vorstellung der Google Kubernetes Engine auf der Google Cloud Platform.

### 17.5.2 Google Kubernetes Engine

Bevor Sie Google Cloud-Dienste nutzen können, benötigen Sie ein Google-Konto. Wenn Sie noch kein Google-Konto angelegt haben, zum Beispiel zur Nutzung von Gmail, dann können Sie das auf der Webseite von Google erledigen. Öffnen Sie die Google-Startseite und wählen dort oben rechts das Steuerelement „Anmelden“ und anschließend „Konto erstellen“. Geben Sie in dem angezeigten Formular die geforderten Daten ein und schließen Sie die Eingabe durch einen Mausklick auf den Button [Konto erstellen] ab.

#### *17.5.2.1 Die Google Cloud Console*

Google stellt für die Verwaltung von Web-Anwendungen eine umfangreiche Web-Admin-Benutzeroberfläche zur Verfügung. Diese nutzen wir hier, um ein einfaches Beispiel-Cluster durch GKE einzurichten. Die Google Cloud Console ist über diese URL erreichbar:

```
1 https://console.cloud.google.com/
```

Wenn Sie noch nicht angemeldet sind, dann werden Sie zunächst auf eine Login-Seite geleitet. Nach erfolgreicher Anmeldung öffnet Sich dann die Startseite der Google Cloud Console.



Wird diese von Ihnen zum ersten Mal gestartet, dann erscheint ein Begrüßungsdialog, auf dem Sie das Land auswählen können. Sie werden auch aufgefordert, die Nutzungsbedingungen zu akzeptieren. Wenn Sie möchten, können Sie zustimmen, dass Sie per E-Mail über Neuigkeiten informiert werden. Beenden Sie diesen Dialog durch den Button [Zustimmen und fortfahren]. Danach wird auf der Begrüßungsseite die Information ausgegeben, dass für Einsteiger eine 12-monatige Testversion genutzt werden kann. Dabei erhält man für diese Zeit ein Guthaben von 300,- $.

**!!! ACHTUNG !!!**

Wenn der Betrag von 300,- $ überzogen wird, dann fallen für Sie Kosten an.

Wenn Sie diese Option nutzen, dann fragt Google anschließend Informationen zu Ihrem Zahlungsprofil, Kundeninformationen, Zahlungsoption und Zahlungsmethode ab. Man wird auch noch informiert, dass nach Ablauf des Testzeitraums keine automatische Gebühr abgebucht wird. Dazu ist ein manuelles Upgrade auf ein kostenpflichtiges Konto nötig.

Mit dem Button [Kostenlose Testversion starten] folgen noch einige Fragen zur Nutzung der Google Cloud Console. Dann wird endlich eine Begrüßungsseite angezeigt und wir können loslegen.

### *17.5.2.2 Erstellen eines Kubernetes-Clusters*

Praktisch alle Aktionen der Google Cloud Console werden über das Navigationsmenü auf der linken Seite des Fensters gesteuert. Falls dieser Bereich nicht angezeigt wird, dann kann er über die Schaltfläche mit den drei waagrechten Linien oben links im Console-Fenster sichtbar gemacht werden (Abb. 17.23):

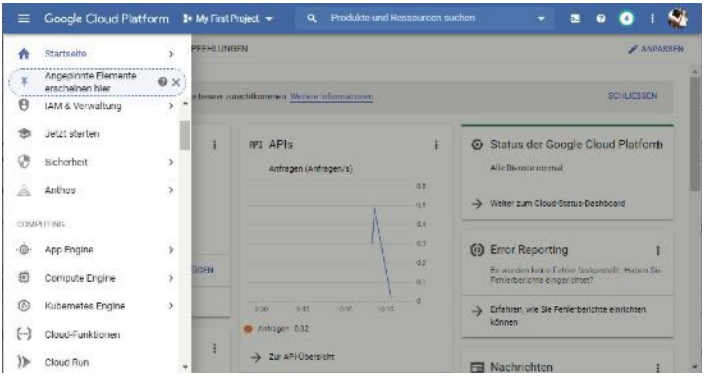



**Abb. 17.23** Navigations-Menü der “Google Cloud Console”

Wir wählen im Navigationsmenü den Menüpunkt KUBERNETES ENGINE und dann aus dem Untermenü den Menüpunkt CLUSTER. Danach wird eine Seite mit dem Kubernetes Cluster Bereich der Google Cloud Console geöffnet (Abb. 17.24):

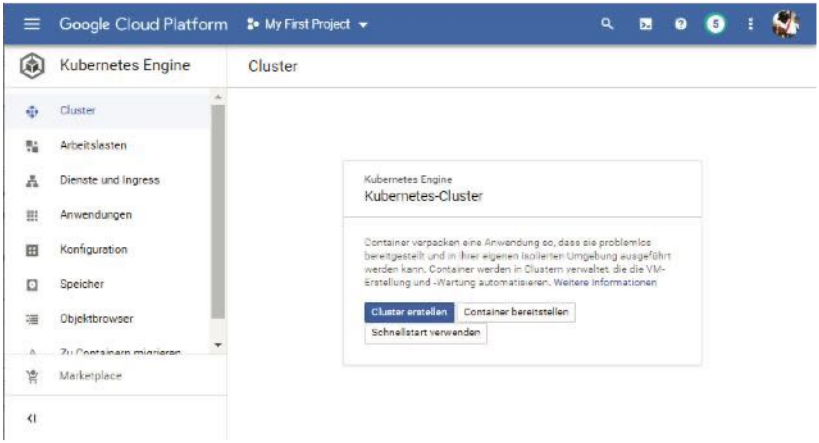



**Abb. 17.24** Die Kubernetes-Cluster-Seite in der „Google Cloud Console“

Durch einen Klick auf den Button [Cluster erstellen] wird ein Wizzard gestartet, der uns durch die Erstellung eines Clusters führt. Es wird zu Beginn die Kubernetes Engine gestartet, darum kann es einen Moment dauern, bis mit der Erstellung des Clusters begonnen werden kann.

Im ersten Schritt geben wir im Fenster die Daten für die „Cluster Grundlagen“ ein. Geben Sie hier einen Namen für das Cluster an. Der Cluster-Name muss aus einer Kombination von Kleinbuchstaben, Ziffern und Bindestrichen zusammengesetzt sein.

Darunter werden zwei Optionen für den Standorttyp angeboten, nämlich *Zonal* und *Regional*. Der Regional-Typ ist neuer und etwas robuster als der Zonal-Typ und die Nodes werden über mehrere Zonen verteilt. Da er aber mehr Ressourcen beansprucht, wählen wir für unser erstes Cluster die Option *Zonal*.

Aus der Drop-Down-Liste *Region* wählen wir eine Region aus, z.B. ‚*europe-west3-a*' steht für die Region Deutschland/Frankfurt.

Zuletzt geben wir an, welche Kubernetes-Version auf dem Master Node ausgeführt werden soll. Als Optionen für die Master-Version steht einmal ‚*Release Kanal*' und ‚*Statische Version*' zur Verfügung.

Aktiviert man die Option *Release Kanal*, dann werden automatische Updates ausgeführt, wenn neue Versionen von Kubernetes verfügbar sind. Wird hier *Statische Version* ausgewählt, dann kann die gewünschte Version von Kubernetes aus der darunterliegenden Drop-Down-Liste ausgewählt werden. Soll später eine neuere Version verwendet werden, dann muss das manuell eingestellt werden.

Im Anschluss sehen Sie einen Screenshot der Seite mit den Cluster-Grundlagen (Abb. 17.25):

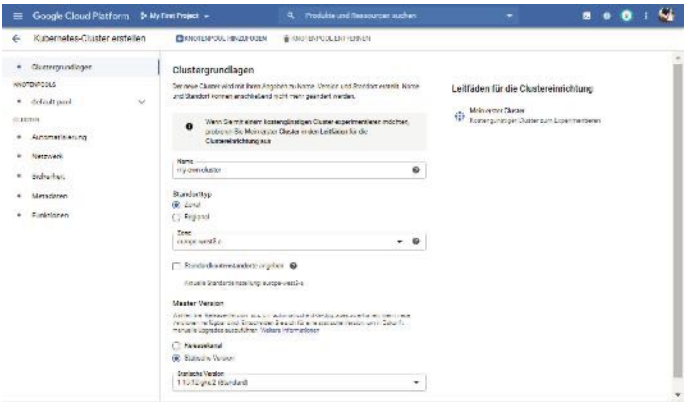



**Abb. 17.25** Die Seite „CLUSTER GRUNDLAGEN" in der „Google Cloud Console"

*KNOTENPOOLS / Default Pool*

Jetzt wollen wir Anzahl und Größe der Nodes für unser Cluster bestimmen. Dazu wählen wir links im Navigationsbereich unter der Sektion KNOTENPOOLS den Eintrag *default-pool* aus.

Rechts wird das Fenster mit den *Knotenpool-Details* angezeigt. Es kann auch hier ein selbstdefinierter Name für einen Knotenpool (Note-Pool) eingetragen werden. In einer Drop-Down-Liste stehen wieder verschiedene Kubernetes-Versionen, die auf den Notes installiert werden, zur Auswahl bereit.

Im Feld *Größe* können Sie die Anzahl der Knoten für diesen Kontenpool bestimmen. Wir haben für das Beispiel die Anzahl 3 gewählt. Was die Angaben zur automatischen Skalierung und zur Automatisierung der Upgrades angeht, übernehmen wir die vorgegebenen Standardeinträge (Abb. 17.26).

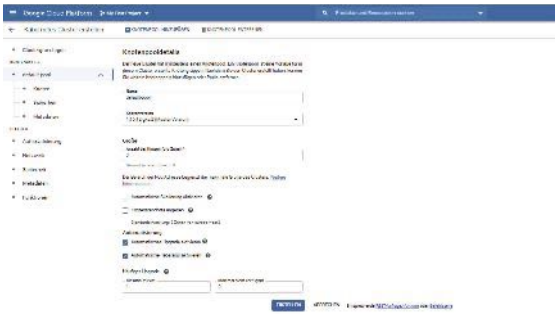


**Abb. 17.26** Die Seite "KNOTENPOOL DETAILS" in der "Google Cloud Console"

*KNOTENPOOLS/Default Pool/Knoten*

Um weitere Angaben zur Konfiguration der Knoten zu machen, wählen wir im Navigationsbereich den Eintrag *Knoten* aus. Hier können Sie die Größe und Leistungsfähigkeit der Nodes bestimmen. Aber man sollte dabei bedenken, dass Größe und Leistungsfähigkeit der Nodes die Kosten für ein Cluster beeinflussen. Wir tragen hier eine Konfiguration ein, welche die Kosten niedrig hält.

Im Feld *Image Type* kann der für die Applikation benötigte Image-Typ aus einer Liste ausgewählt werden. Wir übernehmen hier den Standardwert ‚Container Optimized OS (COS)'.

Bei der Angabe des Maschinentyps wählen wir im Feld *Reihe* ‚E2' und im Feld *Maschinentyp* ‚e2medium'.

Als Bootlaufwerktyp lassen wir die Standardvorgabe ‚Nichtflüchtiger Standardspeicher' und die Größe des Bootlaufwerks reduzieren wir von 100 GB auf 32 GB.

Bei den restlichen Feldern übernehmen wir die eingetragenen Vorgabewerte.

*KNOTENPOOLS/Default Pool/Sicherheit & Metadaten*

In den Sektionen Netzwerk, Sicherheit und Metadaten, die sich unter der Sektion Default Pool befinden, lassen wir die Angaben unverändert.

*CLUSTER/Automatisierung*

Im Fenster Automatisierung unter der Sektion CLUSTER deaktivieren wir die Kontrollkästchen ‚Vertikales POD-Autoscaling' und ‚Automatische Knotenbereitstellung' (Abb. 17.27).

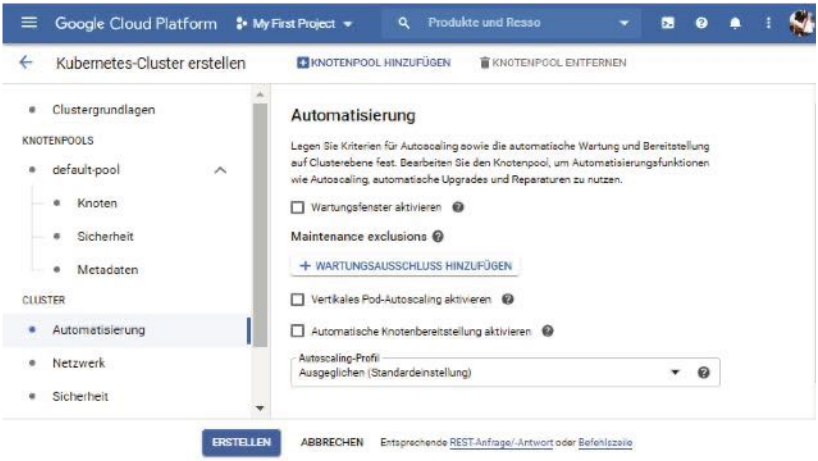



**Abb. 17.27** Die Seite „CLUSTER AUTOMATISIERUNG" in der „Google Cloud Console"

*CLUSTER/Funktionen*

Im Fenster *Funktionen* unter der Sektion CLUSTER wählen wir ‚Kubernetes Engine Monitoring aktivieren' ab, um die dafür eventuell anfallenden Kosten für unsere Übungsbeispiele zu vermeiden (Abb. 17.28).

Durch ‚Kubernetes Engine Monitoring' werden Messwerte, Logs und Ereignisse aus Infrastruktur, Anwendungen und Diensten in Kubernetes-Pods und Clustern zusammengefasst. Dadurch werden detaillierte Informationen zum Verhalten von Applikationen bereitgestellt.

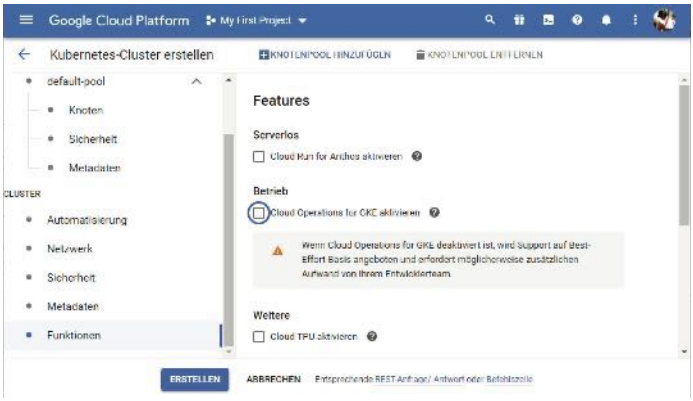



**Abb. 17.28** Die Seite „CLUSTER FUNKTIONEN" in der „Google Cloud Console"

*CLUSTER/Netzwerk & Sicherheit & Metadaten*

In den Sektionen *Netzwerk, Sicherheit und Metadaten* führen wir keine Änderungen der Standard-Eintragungen durch.

Nachdem wir alle notwendigen Angaben zu unserem Cluster eingetragen haben, können wir es erstellen lassen. Dazu klicken wir auf den Button [Erstellen], der am unteren Rand eines jeden Konfigurationsfensters eingeblendet wird.

### *17.5.2.3 Das neue Kubernetes-Cluster untersuchen*

Bevor wir uns mit dem neu erstellten Kubernetes-Cluster verbinden, wollen wir einen kurzen Blick auf seine aktuellen Eigenschaften werfen.

Dazu müssen wir natürlich an der Google Cloud Console angemeldet sein und die Cluster-Seite aus dem Bereich Kubernetes Engine muss aktiv sein. Wenn bereits Cluster erstellt worden sind, dann wird auf dieser Seite eine Liste der verfügbaren Cluster angezeigt. Wahrscheinlich besteht diese Liste bei Ihnen, so wie im folgenden Screenshot, aus nur einem Element (Abb. 17.29):


**Abb. 17.29** Liste der verfügbaren Cluster in der „Google Cloud Console“

Durch einen Mausklick auf den Namen des Clusters in der ersten Spalte wird eine Seite mit den Detail-Informationen zu diesem Cluster geöffnet (Abb. 17.30):

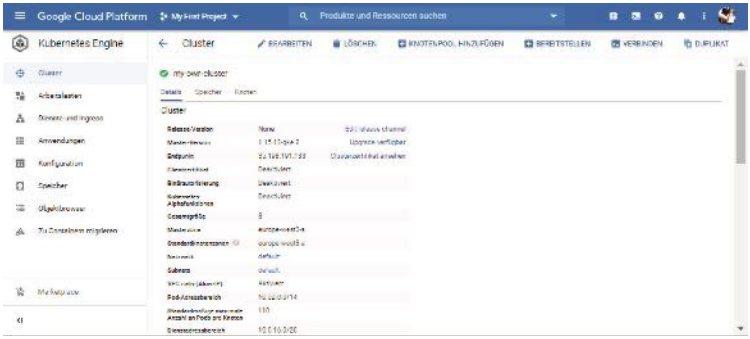



**Abb. 17.30** Cluster-Details in der “Google Cloud Console”

Zusätzlich zum Register „DETAILS“ stehen auf dieser Seite noch zwei weitere Register zur Auswahl, nämlich das Register „SPEICHER“ und das Register „KNOTEN“.

Aktivieren Sie das Register „Knoten“ (Nodes), um eine Liste der Nodes zu erhalten, die zu diesem Cluster gehören (Abb. 17.31).

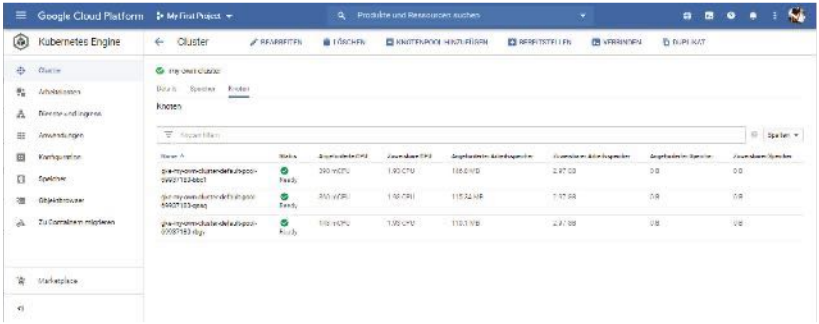



**Abb. 17.31** Node-Liste in der “Google Cloud Console”

Klickt man hier wiederum auf den Namen eines Nodes, so erhält man weitere Detail-Informationen zu diesem Node.

Am besten Sie klicken sich einfach durch diese Seiten, um mit deren Struktur vertraut zu werden. Sie verschaffen sich dabei zusätzlich einen Überblick über die Eigenschaften und Einstellungen des Clusters und seiner Nodes.



### *17.5.2.4 Mit dem Cluster verbinden*

Nachdem wir ein Kubernetes-Cluster mit einem Manager Node und drei Worker Nodes erstellt haben, ist es an der Zeit, auf diesem Cluster eine Anwendung bereitzustellen.

Dazu aktivieren Sie die Cluster-Seite aus dem Bereich Kubernetes Engine. In der rechten Spalte der Node-Liste wird für jeden Node ein Button mit der Beschriftung [VERBINDEN] angeboten (siehe Abb. 17.32).

Wir klicken auf diesen Butten und es erscheint das folgende Dialogfenster:

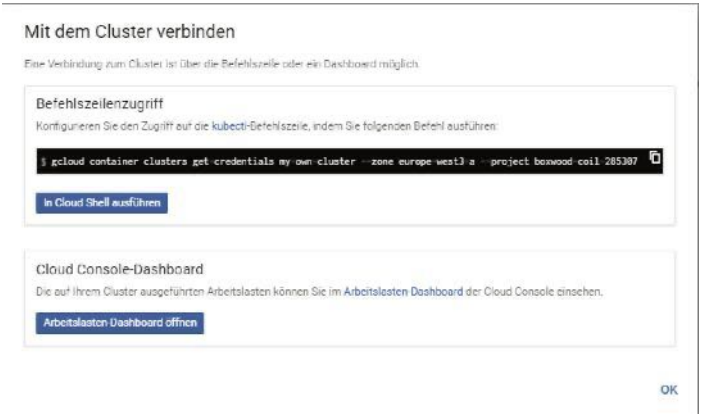



**Abb. 17.32** Dialog „MIT DEM CLUSTER VERBINDEN“ in der „Google Cloud Console“

In diesem Dialog wird ein `gcloud`-Kommando angezeigt, welches in einer GCloud Console eingegeben werden muss, um sich mit dem Cluster zu verbinden. Das Kommando kann man von diesem Dialog aus in die Zwischenablage kopieren, um es anschließend in einer Shell wieder einzufügen und auszuführen.

Die Cloud Shell stellt übrigens ein Online-Terminal bereit, welches wir für die nächsten Schritte in unserem Beispiel nutzen werden.

Sie haben aber auch die Möglichkeit, das Google Cloud SDK über den folgenden Link auf Ihren Arbeitscomputer herunterzuladen und zu installieren.

https://cloud.google.com/sdk

Damit werden unter anderem die Kommandozeilen Utilities `gcloud` und `kubectl` installiert. Sie können in diesem Fall das Kommando zum Verbinden mit dem Cluster über die Zwischenablage in ein lokales Kommando-Fenster kopieren und ausführen.

Im Dialogfenster „Mit dem Cluster verbinden“ befindet sich unter der Anzeige des Konsole-Kommandos ein Button mit dem Label [In Cloud Shell ausführen] (siehe Abb. 17.32). Ein Klick darauf bietet die wirklich komfortable Möglichkeit, ein Online-Terminal zu starten und das Kommando direkt dort auszuführen. War die Verbindung mit dem Cluster erfolgreich, dann können wir auch schon ein Erstes `kubectl`-Kommandos eingeben und wir lassen uns die Liste der Nodes anzeigen (Abb. 17.33):

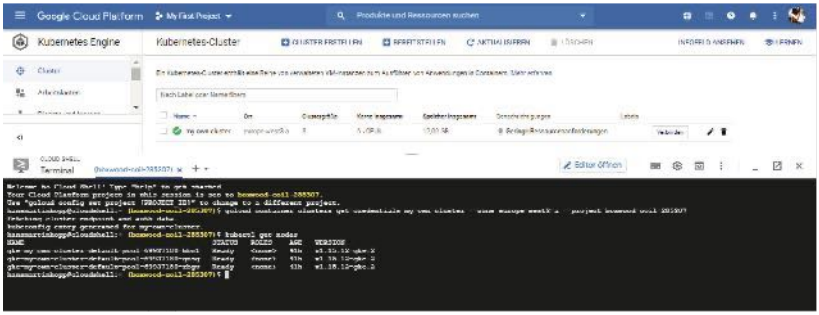



**Abb. 17.33** Das Online-Terminal der “Google Cloud Console”

Das Online-Terminal kann man übrigens jederzeit über das Symbol „Cloud Shell aktivieren“ im Kopfbereich der Google Cloud Console wieder zuschalten (Abb. 17.34).

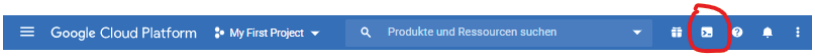



**Abb. 17.34** Das Symbol Cloud Shell aktivieren in der „Google Cloud Console“

### *17.5.2.5 Manifest-Dateien für das Deployment anlegen*

Um das Deployment zu erstellen, soll nun wieder der deklarative Ansatz mit Manifest-Dateien eingesetzt werden. Jetzt befinden wir uns allerdings nicht mehr lokal auf unserem Arbeitsrechner, sondern wir arbeiten mit einer virtuellen Maschine in einer Cloud, auf die wir über ein Terminal zugreifen. Dort müssen sich dann eben auch die YAML-Dateien befinden.

Für den Fall, dass wir Manifest-Dateien verwenden wollen, die wir schon erstellt haben, die sich aber auf unserer lokalen Festplatte befinden, müssen wir diese aber nicht noch einmal abtippten. Die Google Cloud-Plattform bietet dafür zum Glück die Funktionalität, um lokale Dateien auf den Manager Node zu übertragen.

Wir werden die Manifest-Dateien `'nginx-app.yaml'` und `'nginx-svc.yaml'` aus den Kapiteln 17.4.5.1 und 17.4.6.1 wieder verwenden, um damit die NGINX-Applikation in der Google Cloud bereitzustellen.

Um eine Datei vom lokalen Rechner hochzuladen, öffnen Sie für das gewünschte Cluster die Cloud Shell. In der Titelleiste der Cloud Shell befindet sich im rechten Bereich ein Symbol mit drei Punkten. Durch Mausklick auf dieses Symbol öffnet sich ein Drop-Down-Menü, das unter anderem den Menüpunkt *Datei hochladen* anbietet. Damit wird ein Datei-Dialog geöffnet, der es erlaubt, die gewünschte Datei auf Ihrem Rechner auszuwählen und in den Node hochzuladen (Abb. 17.35):

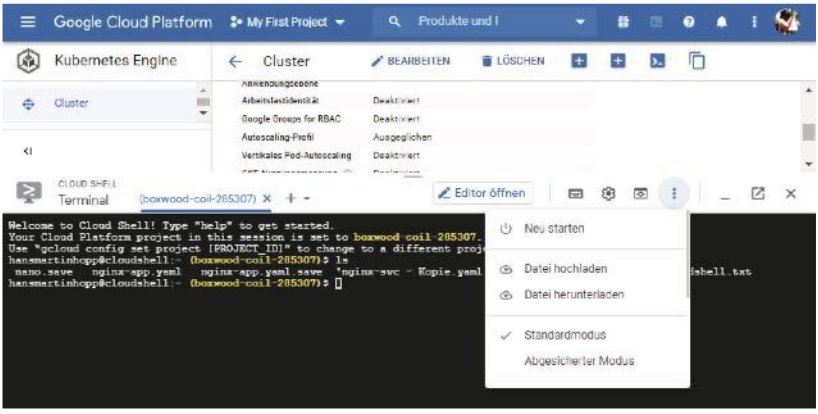



**Abb. 17.35** Lokale Datei in die Cloud mit der „Google Cloud Console" hochladen

Wählen wir jetzt unsere Manifest-Dateien über den angezeigten Dialog „Datei öffnen“ aus und klicken dann den Button [Öffnen], um diese in die Cloud hochzuladen (Abb. 17.36):

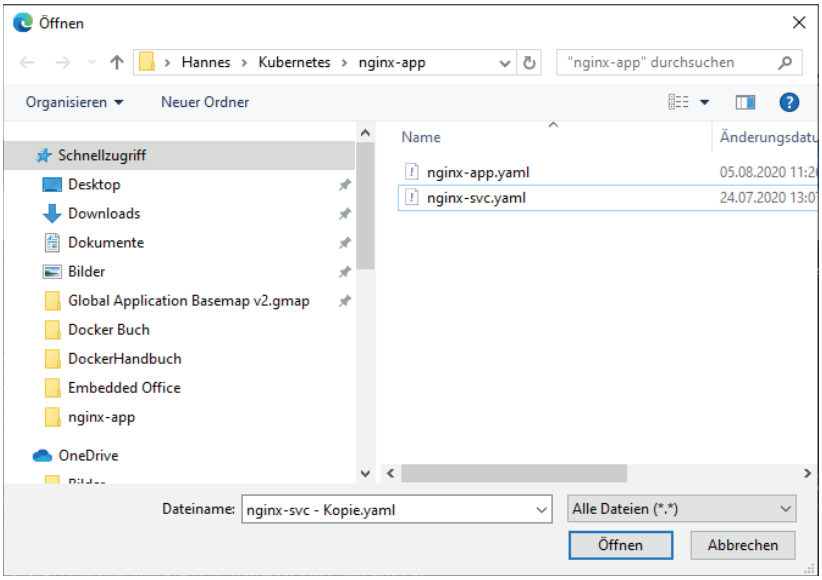



**Abb. 17.36** Die Manifest-Dateien im Dialog „Datei öffnen“

Mit dem Linux-Kommando `ls` können Sie in der Cloud Shell nachsehen, ob die Dateien auch tatsächlich vorhanden sind.

Wenn wir diese Dateien für das Deployment in der Shell noch ändern wollen, dann können wir sie über das Terminal-Fenster weiter editieren.

Zum komfortablen Editieren von Dateien innerhalb eines Nodes gibt es einen Online Editor. Den startet man über den Button [Editor öffnen] am oberen Rand des Terminal-Fensters. Im linken Bereich des Editor-Fensters kann man Dateien markieren. Rechts ist das Bearbeitungsfenster, in welchem die Änderungen am Datei-Inhalt vorgenommen werden können. Über die Menüleiste kann auf die bei Editieren übli-

chen Menüpunkte zugegriffen werden (Datei öffnen, Datei speichern, kopieren, einfügen etc.) (Abb. 17.37):

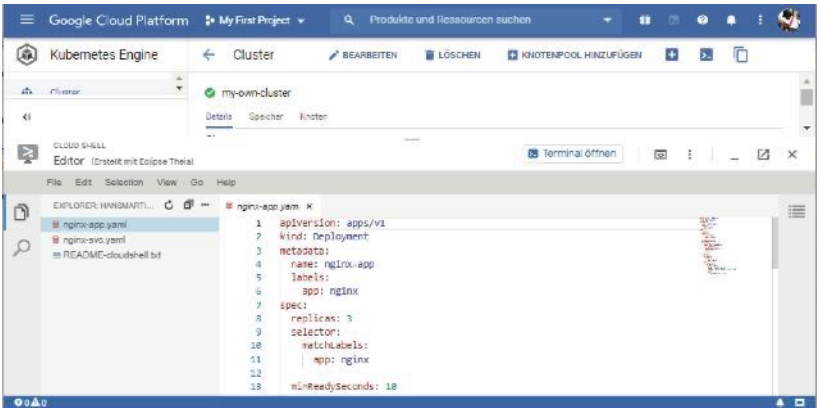



**Abb. 17.37** Der Online Editor aus der "Google Cloud Console"

Alternativ können Sie natürlich auch einen anderen Editor, wie zum Beispiel `vi` oder `nano` verwenden, um Dateien zu erstellen und zu bearbeiten. Diese beiden Editoren sind in den Nodes bereits installiert.

Mit der Schaltfläche [Terminal öffnen], welche die Schaltfläche [Editor öffnen] ersetzt hat, schließt man den Editor und gelangt zurück zum Terminal-Fenster.

Die Datei `'nginx-svc.yaml'` müssen wir noch etwas modifizieren, damit der Service als Load Balancer arbeitet und damit wir von außen über eine externe IP auf unsere Anwendung zugreifen können.

```
Datei 'nginx-svc.yaml'

apiVersion: v1

kind: Service

metadata:
  name: nginx-svc
  labels:
```

```
    app: nginx

spec:
  type: LoadBalancer
  loadBalancerIP: ""
  ports:
  - protocol: "TCP"
    nodePort: 32580
    port: 80
  selector:
    app: nginx
```

### *17.5.2.6 Das Deployment erzeugen*

Mit diesen Manifest-Dateien im Manager Node können wir jetzt wie gewohnt das Deployment und den Service erzeugen.

Erst erzeugen wir das Deployment und sehen uns die Informationen dazu an (Abb. 17.38):

```
$ kubectl create -f nginx-app.yaml
$ kubectl get deployment nginx-app
$ kubectl get pods
```

```
hansmartinhopp@cloudshell:~ (boxwood-coil-285307)$
hansmartinhopp@cloudshell:~ (boxwood-coil-285307)$ kubectl create -f nginx-app.yaml
deployment.apps/nginx-app created
hansmartinhopp@cloudshell:~ (boxwood-coil-285307)$ kubectl get deployment nginx-app
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
nginx-app   3/3     3            0           10s
hansmartinhopp@cloudshell:~ (boxwood-coil-285307)$ kubectl get pods
NAME                         READY   STATUS    RESTARTS   AGE
nginx-app-68c7f5464c-fptpn   1/1     Running   0          53s
nginx-app-68c7f5464c-svwd5   1/1     Running   0          53s
nginx-app-68c7f5464c-z4zf6   1/1     Running   0          53s
hansmartinhopp@cloudshell:~ (boxwood-coil-285307)$
```

**Abb. 17.38** Deployment in der „Google Cloud Console“ erstellen und prüfen

Dann erstellen wir den Service und lassen uns die Liste der Services ausgeben (Abb. 17.39):

```
$ kubectl create -f nginx-svc.yaml
$ kubectl get svc
```

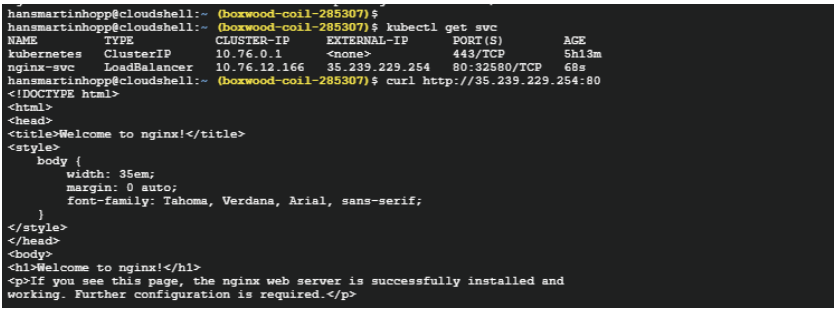



**Abb. 17.39** Service in der „Google Cloud Console“ erstellen und prüfen

Es kann passieren, dass bei der Abfrage der Service-Informationen in der Spalte EXTERNAL-IP für den Service `'nginx-svc'` die Angabe `<pending>` steht. Das bedeutet, dass die Externe Schnittstelle noch nicht bereit ist und wir müssen noch einen Moment warten. Bei einem späteren Aufruf von `kubectl get svc` sollte dann eine IP angezeigt werden. Mit dieser IP können wir auf die Pods unserer Anwendung über die Serviceschnittstelle zugreifen.

Der Zugriff auf diese IP wird im obigen Screenshot mithilfe eines `curl`-Kommandos getestet.

### *17.5.2.7 Das Deployment untersuchen*

Detaillierte Informationen über Deployments in einem Cluster werden auch von der Google Cloud Console angeboten.

Dazu wählt man im Navigationsmenü den Menüpunkt KUBERNETES ENGINE und dann aus dem Untermenü den Menüpunkt ARBEITSLASTEN (= Workload). Danach wird eine Seite mit dem Kubernetes-Bereich zur Untersuchung der Workloads in der Google Cloud Console geöffnet. Die aktuell verfügbaren Deployments werden in einer Liste angezeigt. Will man genauere Informationen zu einem Deployment, dann klickt man mit der Maus auf dessen Namen. Damit wird die Seite mit den weiterführenden Informationen für dieses Deployment geöffnet.

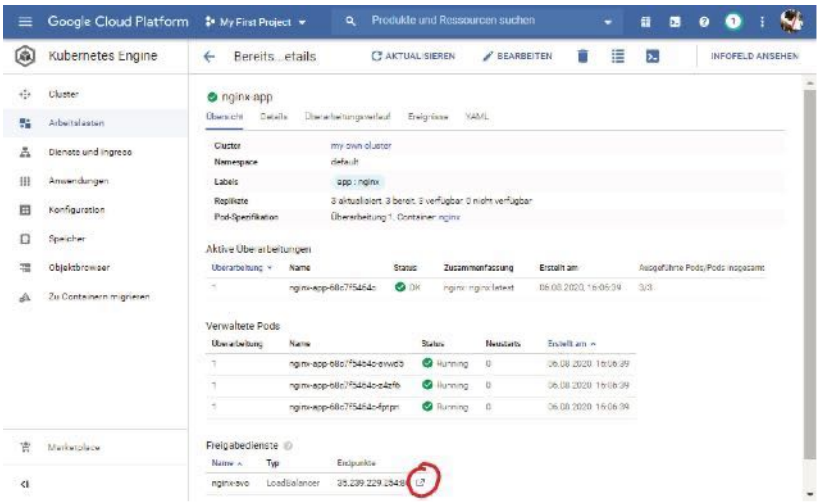



**Abb. 17.40** Die Seite „ARBEITSLASTEN“ in der „Google Cloud Console“

Am Ende dieser Seite befindet sich der Bereich „Freigabedienste“. Dort erhalten Sie Zugriff auf die vorhandenen Services. In der Liste ist bei uns der Eintrag für den Service `nginx-svc` sichtbar. Dort taucht in der rechten Spalte die IP des Endpunktes für den Externen Zugriff auf.

Rechts neben diesem Eintrag befindet sich eine Art Rechtecksymbol mit einem Pfeil (siehe roter Kreis). Ein Mausklick auf dieses Element öffnet die zugehörige Webseite direkt im Internet-Browser und es erscheint wieder die bekannte Begrüßungsseite aus dem NGINX Image.



Man kann die IP des Endpunktes natürlich auch von Hand in die Adresszeile eines Browsers eintippen, um die Startseite einer Web-Anwendung aufzurufen.

#### *17.5.2.8 Löschen des Clusters*

Damit nicht irgendwann unnötige Kosten anfallen, wird das Cluster noch gelöscht. Das geht aber relativ einfach.

Aktivieren Sie die CLUSTER-Seite aus dem Bereich KUBERNETES ENGINE. In der dort angezeigten Cluster-Liste befindet sich in der rechten Spalte bei jedem Eintrag ein kleines Mülleimer-Symbol (siehe Abb. 17.29 im Kapitel 17.5.2.3). Mit einem Mausklick auf dieses Symbol wird das zugehörige Cluster wieder gelöscht. Das dauert allerdings eine gewisse Zeit.

#### *17.5.2.9 Übungsaufgabe: Die Applikation Telefon-App bereitstellen*

Im Kapitel 4.9 haben wir ein Image für eine einfache PHP-Anwendung erstellt, mit der wir eine Telefonnummer in einer Liste suchen und anzeigen können. Dieses Image haben wir im Rahmen dieses Beispiels bei Docker Hub veröffentlicht.

In dieser Übung soll diese Anwendung mithilfe von GKE bereitgestellt werden.

Die Manifest-Dateien erstellen wir zunächst lokal in einem neuen Verzeichnis unterhalb des Verzeichnisses Kubernetes mit dem Namen `telefon-app`:

```
1  <USER_DIR>\Kubernetes\telefon-app
```

Dort erstellen wir die beiden Manifest-Dateien mit den Namen `'telefon-app.yaml'` und `'telefonsvc.yaml'`.

Als Name des Deployments verwenden wir `'telefon-app'`. Die Sektionen erhalten das Label `'telefon'`. Der Container soll ebenfalls den

Namen `'telefon'` erhalten. Bei diesem Beispiel sollen für das Deployment nur 2 Replikate instanziiert werden.

Als Image wird jetzt der Name Ihres Images eingetragen, den Sie bei Docker Hub verwendet haben. Dem Image-Namen muss Ihre Docker ID, die Sie auch zum Anmelden an Docker Hub angeben müssen, durch einen Schrägstrich getrennt vorangestellt werden. Hier ein Beispiel dazu:

```
1  image: monikamuster/telefon-php
```

Der Service erhält den Namen `'telefon-svc'`. Die Sektionen erhalten auch hier das Label `'telefon'`.

Die restlichen Einträge übernehmen wir von unserem NGINX Deployment.

Nun erstellen wir ein Kubernetes Cluster mit dem Namen `'telefon-cluster'`. Der Knotenpool soll bei dieser Übung 2 Knoten bereitstellen. Die restlichen Einstellungen sind wieder die gleichen wie wir Sie in Kapitel 17.5.2.2 angegeben haben.

Wir übertragen anschließend die Manifest-Dateien in das Cluster und erstellen das Deployment und den Service.

Abschließend wird das Deployment noch getestet.

***Lösung:***

Die Manifest-Dateien:

```
Datei 'telefon-app.yaml'
apiVersion: apps/v1
kind: Deployment
metadata:
  name: telefon-app
  labels:
    app: telefon
spec:
  replicas: 2
  selector:
    matchLabels:
      app: telefon

  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1

  template:
    metadata:
      labels:
        app: telefon
    spec:
      containers:
      - name: telefon
           image: <DOCKER_ID>/telefon-php
        ports:
        - containerPort: 80
```

```
Datei 'telefon-svc.yaml'
apiVersion: v1

kind: Service

metadata:
  name: telefon-svc
  namespace: default
  labels:
    app: telefon
```

```
11
12 spec:
13   type: LoadBalancer
14   loadBalancerIP: ""
15   ports:
16   - protocol: "TCP"
17   - nodePort: 32580
18     port: 80
19   selector:
20     app: telefon
```

Verbinden Sie sich mit dem Cluster und öffnen das Online-Terminal (siehe Kapitel 17.5.2.4).

Die beiden Dateien werden mithilfe der Google Cloud Console in den Manager Node des Clusters kopiert (siehe Kapitel 17.5.2.5).

Das Deployment erzeugen:

```
1 $ kubectl create -f telefon-app.yaml
```

Den Service erzeugen:

```
1 $ kubectl create -f telefon-svc.yaml
```

Die EXTERNE IP des Service herausfinden:

```
1 $ kubectl get svc
```

oder auf der Seite ARBEITSLASTEN im dem Bereich KUBERNETES ENGINE der Google Cloud Console (siehe Abb. 17.40 in Kapitel 17.5.2.7).



Wird diese IP in den Browser eingegeben, dann sollte die Webseite unserer Telefon-App erscheinen.

**!!! Vergessen Sie nicht zum Schluss das Cluster wieder zu löschen !!!**

Kapitel 18

# Wie geht es weiter?

Bei vielen Fachbüchern findet man zum Abschluss an dieser Stelle so Aussagen wie „Herzlichen Glückwunsch, Sie haben es geschafft!". Für ein Buch zum Thema Docker finde ich das allerdings nicht so angebracht. Besser wäre da dann schon die Aussage „Herzlichen Glückwunsch, jetzt sind Sie an der Reihe".

Damit die Anwendung des bisher Gelernten in Fleisch und Blut übergeht und in Zukunft effektiv angewendet werden kann, sind weitere Aktivitäten Ihrerseits nötig.

Da ist zum einen die Übung. Praxis bekommt man nur durch regelmäßiges Praktizieren. Darum möchte ich Sie dazu ermuntern, dass Sie kreativ werden, Ideen für neue Anwendungsfälle entwickeln und diese dann mit Docker realisieren.

Mein zweiter Ratschlag ist: „Bleiben Sie am Ball". Wie bereits am Anfang dieses Buches bemerkt wurde, gilt in der Welt des Cloud-Computing mit Docker und Kubernetes: „Ständige Veränderung ist die neue Normalität!" Wenn man sich da nicht laufend weiterbildet und über Änderungen und Neuerungen informiert ist, dann wird man recht schnell von der Entwicklung abgehängt. Werfen Sie deshalb regelmäßig einen Blick in die Online-Dokumente von Docker, Kubernetes, Google, Amazon und ähnliche Web-Portale, um stets über die aktuellen Veränderungen informiert zu bleiben.

Damit wünsche ich Ihnen zum Schluss: „Willkommen in der abenteuerlichen Welt von Docker und viel Spaß!"

Kapitel 19

# Anhang

## 19.1 MAC-OS Installation von Docker

### 19.1.1 Docker Desktop für MAC-OS installieren

#### *19.1.1.1 Systemvoraussetzungen*

Minimum-Voraussetzung für eine erfolgreiche Docker-Installation unter MAC-OS ist eine OS-Version größer 10.13, zurzeit also macOS Catalina, macOS Mojave oder macOS High Sierra. Die MAC Hardware muss Baujahr 2010 oder größer sein mit Intel Hardware Support für MMU (Memory Management Unit)-Virtualisierung und EPT (Extended Page tables).

Um dies herauszufinden, können Sie in einem Terminal das folgende Kommando eingeben:

```
1 sysctl kern.hv_support
```

Als Ergebnis muss dabei Folgendes ausgegeben werden:

```
1 kern.hv_support: 1
```

Es muss mindestens 4 GB RAM installiert sein.

VirtualBox-Versionen, die älter als Version 4.3.30 sind, dürfen nicht installiert sein. Das führt zu Problemen mit Docker Desktop.



#### *19.1.1.2 Download des Installationsprogramms*

Das Docker-Installationsprogramm kann über die Docker-Homepage heruntergeladen werden.

Hier der Link auf diese Seite:

https://www.docker.com/

Der folgende Screenshot zeigt die Startseite von Docker. Diese Seite wird bei Ihnen zu einem späteren Zeitpunkt wahrscheinlich wieder etwas anders aussehen (Abb. 19.1):

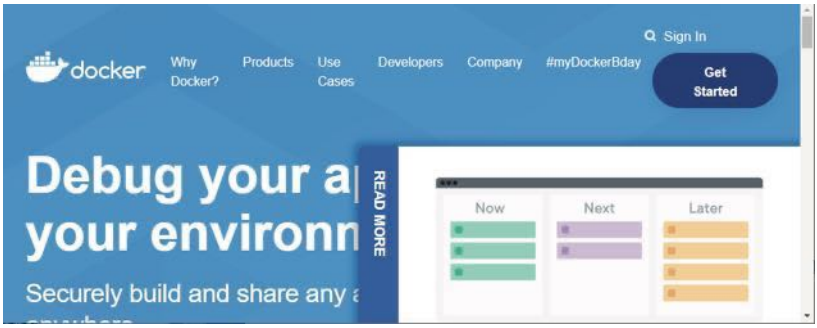



**Abb. 19.1** Die Startseite von Docker im Internet

Bewegen Sie auf dieser Seite den Mauszeiger über den Menütext [PRODUCTS]. Dadurch öffnet sich das folgende Untermenü (Abb. 19.2):

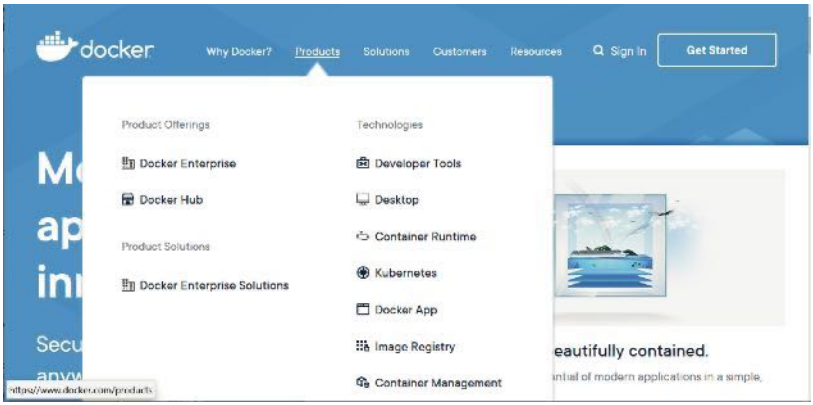



**Abb. 19.2** Docker-Produkte auf der Homepage

In diesem Untermenü den Menüpunkt [Desktop] mit der Maus anklicken. Es wird zur Download-Seite von Docker Desktop weitergeleitet (Abb. 19.3):


**Abb. 19.3** Download-Seite von Docker Desktop für Mac und Windows

Falls Sie noch nicht registriert sind, starten Sie die Registrierung auf dieser Seite über den Button [Sign in]. Folgen Sie auf den nächsten Seiten den Anweisungen, um Ihre Registrierung durchzuführen. Sie geben dabei Ihre Docker ID an und ein Passwort. Bitte merken Sie sich diese Angaben gut. Wir werden sie im Laufe der weiteren Praxisübungen immer wieder benötigen.

Falls Sie schon registriert sind, erscheint die folgende Anmeldeseite. Geben sie hier Ihre Docker ID und das Passwort ein und aktivieren Sie die Schaltfläche [Sign In], um sich anzumelden.

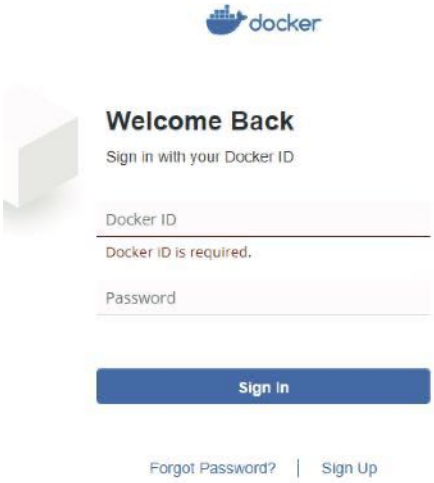



**Abb. 19.4** Docker-Anmeldeseite

Nach erfolgreicher Anmeldung landet man auf der Webseite „Get Docker Desktop for MAC“ (Abb. 19.5):

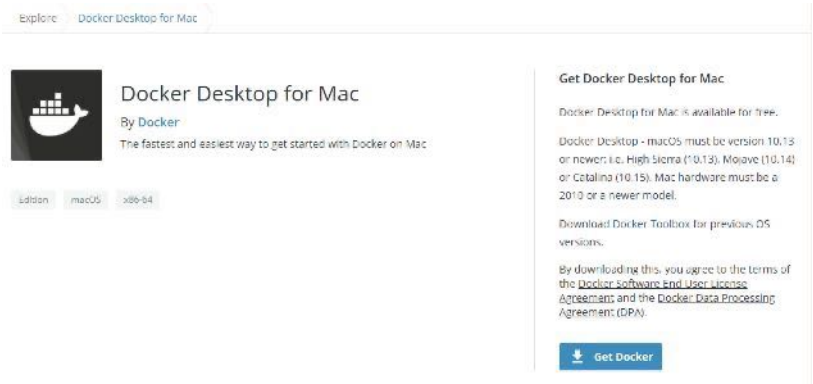



**Abb. 19.5** Docker-Seite , Get Docker Desktop for MAC ‘

Wir aktivieren den Button [Get Docker], um den Download zu starten.

Daraufhin wird die Datei ‚Docker.dmg' heruntergeladen.

#### *19.1.1.3 Installation von Docker Desktop*

Nach vollständigem Download führen Sie einen Doppelklick auf die heruntergeladene Datei aus, um das Disk Image als DMG-Laufwerk zu mounten. Das (virtuelle) Laufwerk wird auf dem Desktop angezeigt. Durch Doppelklick auf das DMG-Laufwerk starten Sie die Installation.

Wie bei MAC-OS üblich, verschieben Sie zur Installation von Docker Desktop das Docker-Symbol per Drag and Drop mit der Maus in den Ordner *„Applications"*. Bitte bestätigen Sie bei der Sicherheitswarnung, dass Sie das Programm wirklich ausführen möchten.

Klicken Sie jetzt auf den Button [Öffnen] und ziehen Sie das Docker-Symbol noch einmal auf das Ordnersymbol *„Applications"*.

Jetzt wird die Installation durchgeführt und alle benötigten Dateien werden kopiert. Ein Fenster zeigt Ihnen den Fortschritt der Installation an. Ist die Installation vollständig, schließt sich dieses Fenster selbständig.

Wir haben es geschafft und sollten jetzt einen ersten Funktionstest durchführen.

Als Erstes starten wir Docker.

Wir öffnen das Launch Pad und suchen dort nach dem Icon der App „Docker".


Durch Doppelklick starten Sie Docker Desktop. Vorher werden Sie aufgefordert, Docker.app durch Eingabe des System-Passworts zu autorisieren.

In der Statusleiste am oberen Bildschirmrand erscheint jetzt das Docker-Symbol.

Durch Mausklick auf dieses Symbol öffnet man das Docker-Menü. Darüber können verschiedene Optionen und Kommandos für Docker ausgewählt werden.

Wir überprüfen jetzt, ob Docker Desktop korrekt gestartet wurde.

#### *19.1.1.4 Test der Installation*

Zum Testen von Docker müssen wir zuerst ein Terminal-Fenster öffnen. Gehen Sie dazu in den Ordner „Dienstprogramme“ des Anwendungsverzeichnisses und suchen dort nach der Terminal.app oder suchen Sie einfach nach dem Begriff „Terminal“ in der Spotlight-Suche (Tastenkombination [cmd] + [Leertaste]).

Starten Sie die App und geben Sie dort in das Kommando-Fenster den folgenden Befehl ein:

```
1 $ docker version
```

Wenn Sie als Ergebnis die Versionsinformationen angezeigt bekommen, dann war die Installation erfolgreich.

Weitere Infos zur Installation von Docker unter MAC-OS gibt es auf den Internet-Seiten mit der Dokumentation von Docker:

https://docs.docker.com/docker-for-mac/install/

## 19.2 Linux-Installation von Docker Engine unter Ubuntu Linux

### 19.2.1 Betriebssystem-Anforderungen

Eine der folgenden Ubuntu-Versionen als 64-Bit-Version ist Voraussetzung für eine erfolgreiche Installation der Docker Engine:

- XENIAL 16.04 (LTS)
- BIONIC 18.04 (LTS)
- EOAN 19.10

Die Docker Engine Community Edition unterstützt nur die folgenden Prozessor-Architekturen:

- X86_64 bzw. AMD64
- ARMHF
- ARM64
- S390x (IBM Z)
- ppc64le (IBM Power)

### 19.2.2 Deinstallation von alten Versionen

Falls auf einem System noch alte Docker-Versionen installiert sein sollten, dann müssen diese zunächst mithilfe des `apt-get`-Kommandos deinstalliert werden.

Die Namen der früheren Docker-Versionen lauten `docker`, `docker.io` oder `docker-engine`.

Um diese Versionen zu entfernen, geben Sie das folgende Kommando ein:

```
$ sudo apt-get remove docker docker-engine docker.io containerd
runc
```

Falls keines dieser Pakete installiert ist, wird Ihnen das von `apt-get` als Ergebnis mitgeteilt.

### 19.2.3 Installation der Docker Engine Community Edition

Das neue Paket der Docker Engine Community Edition hat die Bezeichnung `docker-ce`.

Im ersten Schritt aktualisieren wir den APT Package Index für unsere Ubuntu-Distribution:

```
$ sudo apt-get update
```

Dann installieren wir Pakete, die notwendig sind, um Repositories über HTTPS zu nutzen:

```
$ sudo apt-get install apt-transport-https ca-certificates \
curl software-properties-common
```

Wir müssen auch noch den offiziellen Docker GPG-Schlüssel zu APT hinzufügen (GPG – Gnu Privacy Guard – ein freies Kryptographie-System).

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
sudo apt-key add -
```

Wir überprüfen noch kurz die letzten acht Zeichen des Fingerprints für diesen Schlüssel:

```
$ sudo apt-key fingerprint 0EBFCD88
```

Jetzt richten wir das Repository ein (im folgenden Beispiel vom Typ `"stable"` für eine X86_64- bzw. AMD64-Architektur):

```
$ sudo add-apt-repository \
"deb [arch=amd64] \
https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable"
```

Bei Installation für andere Architekturen ersetzen Sie die Zeichenkette `"amd64"` aus dem obigen Beispiel durch eine der folgenden Angaben:

| | |
|---|---|
| Bei ARMhf (hardware floating point instructions) | „armhf“ |
| Bei ARM64 | „arm64“ |
| Bei IBM Power PC 64 little-endian | „ppc64le“ |
| Bei IBM‘s System/390 Servern (IBM Z) | „s390x“ |

Jetzt sind wir soweit – wir können die Docker Community Edition installieren.

Zur Sicherheit aktualisieren wir noch einmal den APT Package Index.

```
sudo apt-get update
```

Dann installieren wir Docker CE (das Beispiel installiert die neueste Version):

```
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Wollen Sie eine bestimmte Version der Docker Engine installieren, dann verwenden Sie die folgende Syntax:

```
$ sudo apt-get install docker-ce=<VERSION> \
docker-ce-cli=<VERSION> containerd.io
```



Um zu prüfen, ob Docker korrekt installiert ist, machen wir einen einfachen Test. Zuerst fragen wir die Version der Docker Engine ab:

```
$ docker version
```

Anschließend starten wir einen Container mit dem `"hello-world"`Image:

```
1 $ docker run hello-world
```

Das Kommando lädt ein Test Image und führt es in einem Container aus. Ist der Container korrekt gestartet worden, dann gibt er einige Informationen aus und beendet sich selbst wieder.

## 19.3 Installation von Docker in einem Linux-Subsystem unter Windows

### 19.3.1 Aktivierung des Windows-Subsystems für Linux

Für alle Linux-Fans unter den Lesern oder wer einmal eine Alternative zur Windows Shell ausprobieren möchte, gibt es für Windows 10 die Möglichkeit ein Linux-Kommandofenster zu nutzen.

Wie schon im Kapitel zur Entwicklungsgeschichte erwähnt, gibt es seit Mai 2019 die Version 2 von WSL, dem Windows-Subsystem für Linux. Damit ist es möglich, verschiedene Linux-Distributionen unter Windows auszuführen. Dort kann man dann auch Docker installieren und nutzen.

Voraussetzung für die Aktivierung von Linux-Distributionen unter Windows WSL ist, dass für Windows der Entwicklermodus eingestellt ist.

Klicken Sie im Windows-Start-Menü auf das Symbol-Einstellungen  , um das Dialogfenster „Windows Einstellungen“ zu öffnen. In diesem Fenster geben Sie im Feld *Einstellungen Suchen* den Begriff „Entwickler“ ein. Anschließend klicken Sie in der Ergebnisliste auf „Einstellungen für Entwickler» (Abb. 19.6):

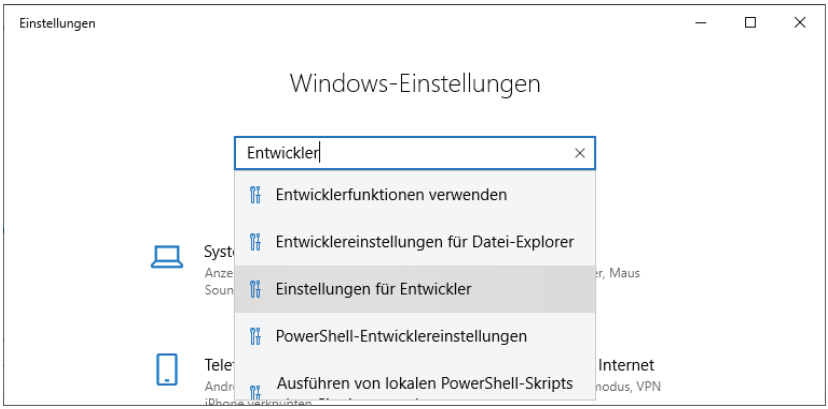



**Abb. 19.6** ENTWICKLEREINSTELLUNG im Dialog „EINSTELLUNGEN“ finden

Jetzt wird das Register FÜR ENTWICKLER gezeigt. Hier aktivieren Sie die Option *Entwicklermodus* (Abb. 19.7):

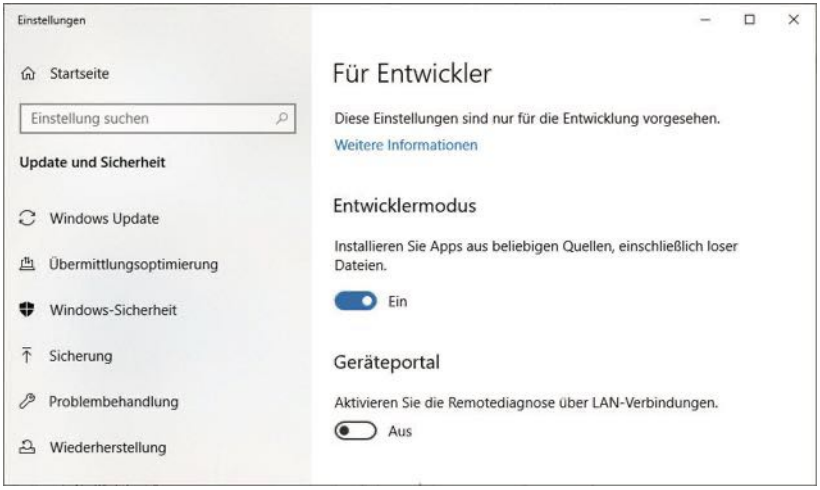



**Abb. 19.7** Einstellungen: Entwicklermodus aktivieren

Anschließend suchen wir noch im Dialog „Windows Einstellungen“ nach „Windows Features“ und wählen aus der Liste den Eintrag Windows-Features aktivieren oder deaktivieren.


**Abb. 19.8** Windows Features im Dialog „Einstellungen“ finden

Nach einer kurzen Wartezeit erscheint das Dialogfenster „Windows Features“. Dort aktivieren Sie wiederum das Kontrollkästchen *Windows-Subsystem für Linux* (Abb. 19.9):

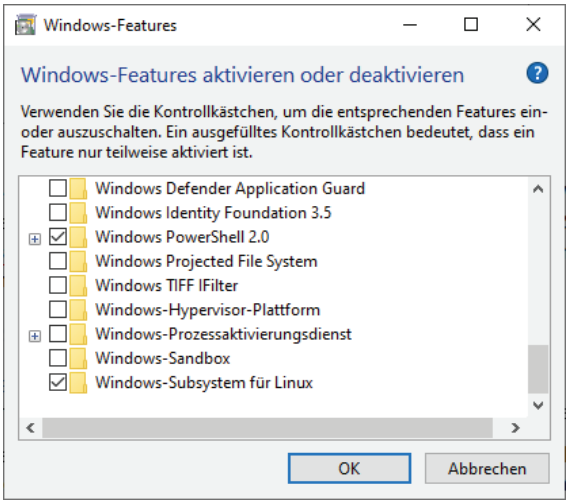



**Abb. 19.9** Windows Features-Dialog

### 19.3.2 Ubuntu-App installieren

Am einfachsten lässt sich Ubuntu aus dem Microsoft Store heraus herunterladen und installieren.

Voraussetzung dafür ist, dass Windows 10 ab Build 16215 aufwärts installiert ist. Auf Ihrem Computer finden Sie die Version des Windows Build auf der „INFO"-Seite heraus.

Wählen Sie im Startmenü wieder EINSTELLUNGEN im Dialogfenster „WINDOW EINSTELLUNGEN", wählen Sie SYSTEM und dort das Register „INFO". Scrollen Sie dann rechts nach unten, bis der Abschnitt „Windows Spezifikationen" angezeigt wird. Hier wird unter anderem der Betriebssystembuild angegeben (Abb. 19.10).

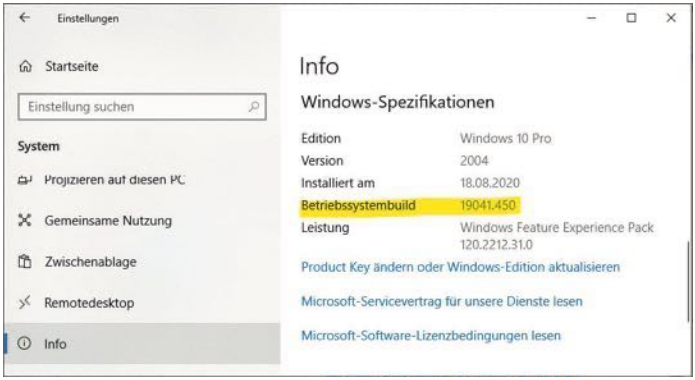



**Abb. 19.10** Windows-Spezifikationen

Wenn bei Ihrem Computer die Systemvoraussetzungen erfüllt sind, dann können Sie die Ubuntu-App über den Microsoft Store bequem installieren.



Starten Sie die App „Microsoft Store" und geben Sie „Linux" als Suchbegriff ein. In der Trefferliste klicken Sie auf „Linux unter Windows ausführen". Sie erhalten eine Auswahl von Apps für verschiedene Linux-Distributionen als Anzeige (Abb. 19.11):


**Abb. 19.11** Linux-Apps in Microsoft Store

Wir wählen Ubuntu aus und klicken dann den Button [Herunterladen], um mit der Installation zu beginnen.

Falls Download und Installation über Microsoft Store nicht möglich sind, kann man Linux-Distributionen in WSL über Powershell-Kommandos herunterladen und installieren.

```
> Invoke-WebRequest -Uri https://aka.ms/wsl-ubuntu-1604 \
-OutFile Ubuntu.appx \
-UseBasicParsing
```

Alternativ kann eine Linux-Distribution auch mithilfe der `curl`-Anweisung heruntergeladen werden. Diese Variante empfiehlt sich, wenn Download und Installation über eine Batch-Datei oder ein Shell-Script durchgeführt werden.

Zu Installation der Linux-Distribution wechseln wir mit der PowerShell in das Verzeichnis, in welches wir die Installationsdatei gerade heruntergeladen haben, und geben den folgenden Befehl ein:

```
> Add-AppxPackage .\Ubuntu.appx
```

### 19.3.3 Initialisierung der Ubuntu-App

Nach der Installation von Ubuntu sind noch ein paar Schritte zur Initialisierung der neuen Distribution notwendig.

Nach erfolgreicher Installation steht Ubuntu im Startmenü bereit und kann gestartet werden. Beim ersten Start öffnet das Kommandofenster und man wird aufgefordert, einen Moment zu warten, bis die Installation fertiggestellt ist. Ist die Installation fertig, können wir einen neuen Benutzernamen und das dazugehörige Kennwort eingeben (Abb. 19.12):

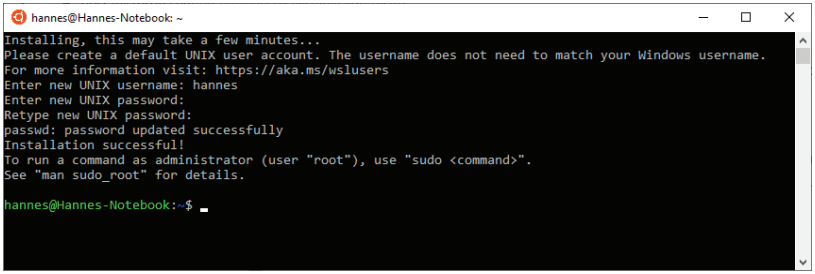



**Abb. 19.12** Das „UBUNTU"-Fenster

### 19.3.4 Docker auf der Ubuntu-App installieren

Im ersten Schritt aktualisieren wir den APT Package Index für unsere Ubuntu-Distribution:

```
1  $ sudo apt-get update
```

Dann installieren wir Pakete, die notwendig sind, um Repositories über HTTPS zu nutzen:

```
1  $ sudo apt-get install apt-transport-https ca-certificates \
2  curl software-properties-common
```

Wir müssen auch noch den offiziellen Docker GPG-Schlüssel zu APT hinzufügen (GPG – Gnu Privacy Guard – ein freies Kryptographiesystem):

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
sudo apt-key add -
```

Wir überprüfen noch kurz die letzten acht Zeichen des Fingerprints für diesen Schlüssel:

```
$ sudo apt-key fingerprint 0EBFCD88
```

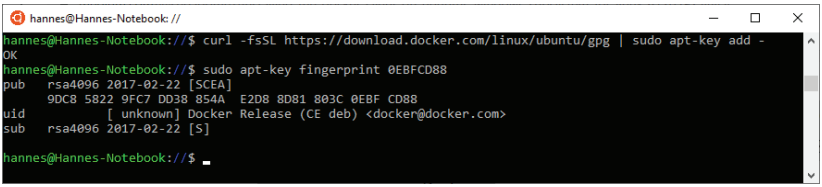



**Abb. 19.13** Zufügen und Prüfung des GPG Keys in der Ubuntu Shell im Windows-Subsystem für Linux

Jetzt richten wir das Repository ein:

```
$ sudo add-apt-repository \
"deb [arch=amd64] \
https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable"
```

Endlich sind wir soweit – wir können die Docker Community Edition installieren.

Zu Sicherheit aktualisieren wir noch einmal den APT Package Index:

```
sudo apt-get update
```

Dann installieren wir Docker CE

```
$ sudo apt-get install docker-ce
```

OK – Docker ist jetzt installiert und wir machen den ersten Test:

```
$ docker run hello-world
```

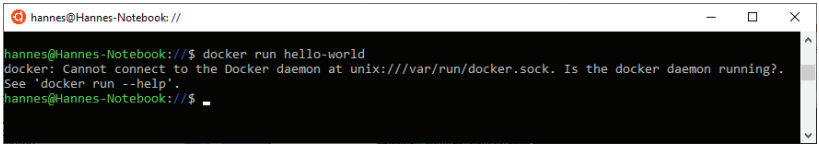



**Abb. 19.14** Erster Test von Docker auf der Ubuntu Shell

Das hat so noch nicht funktioniert? Warum nicht? Es läuft hier tatsächlich keine Docker Daemon. Wir müssen dem Docker Client mitteilen, wo sich der Docker Host befindet. Dafür gibt es für das `docker`-Kommando den Parameter –H:

```
1  $ docker -H localhost:2375 run hello-world
```

ACHTUNG: Damit das funktioniert, müssen wir aber noch sicherstellen, dass die Veröffentlichung des Windows Daemon-Ports aktiviert ist.

Klicken Sie auf das Docker-Symbol in der Task-Leiste, um ein Kontext-Menü zu öffnen. Hier wählen Sie den Menüpunkt SETTINGS:

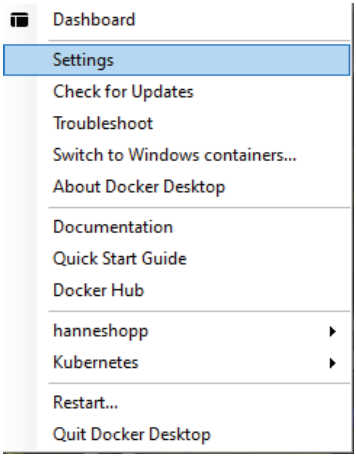



**Abb. 19.15** Das Docker Desktop Kontext Menü „Settings“

Im Dialog-Fenster „SETTINGS“ wählen Sie das Register [GENERAL]. Auf dieser Seite muss das Kontrollkästchen *Expose daemon on tcp://localhost:2375 without TLS* aktiviert sein (Abb. 19.16):

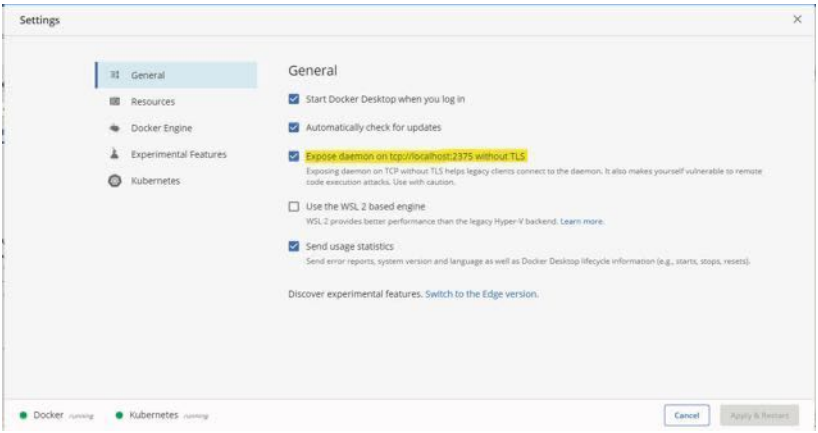



**Abb. 19.16** Docker General Setting: Port für Windows Daemon veröffentlichen

Falls Sie keine Lust haben, den Localhost bei jedem Aufruf von Docker-Kommandos anzugeben, dann können Sie für „Localhost“ eine Systemvariable mit dem Namen `DOCKER_HOST` einrichten und die Angabe dort speichern:

```
1 export DOCKER_HOST='tcp://localhost:2375'
```

Wenn wir jetzt aber die Shell beenden, dann muss die Systemvariable nach dem nächsten Neustart wieder angelegt werden. Um diese Systemvariable dauerhaft einrichten zu können, müssen wir nur in die Konfigurationsdatei `.bash_profile` einen Eintrag hinzufügen. Das geht am schnellsten mit dem folgenden Kommando:

```
1 $ echo "export DOCKER_HOST='tcp://localhost:2375'" >> ~/.bash_
2 profile
```

Beenden Sie das Ubuntu Shell-Fenster und starten Sie es erneut. Geben Sie noch einmal das folgende Docker-Kommando ein:

```
1 $ docker run hello-world
```

Wenn Sie die folgende Ausgabe erhalten, haben Sie es geschafft – Docker kann jetzt auch unter Windows aus einer Ubuntu Shell heraus bedient werden (Abb. 19.17):

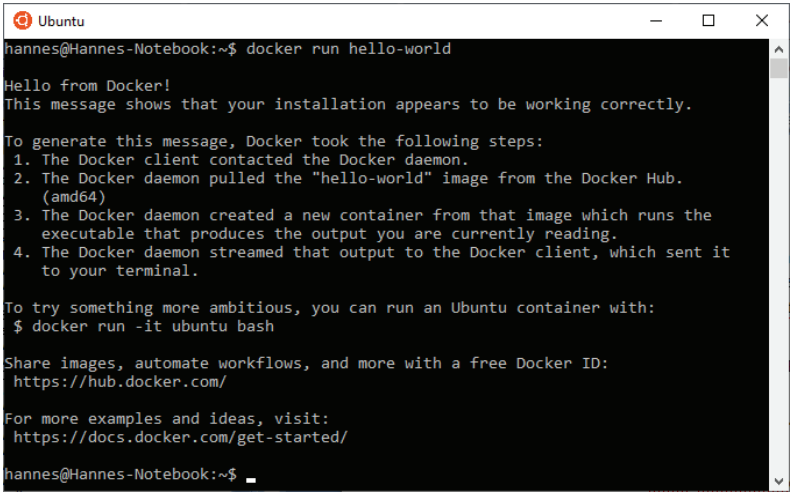



**Abb. 19.17** Der Hello World Container in der Ubuntu-App

## 19.4 Installation von docker-machine

Je nach installierter Version von Docker Engine bzw. von Docker Desktop kann es sein, dass Docker Machine bereits mit installiert wurde oder auch nicht.

Falls Docker Machine auf Ihrer Plattform nicht installiert sein sollte, bieten wir Ihnen in diesem Kapitel für das jeweilige System die Anleitungen zum Nachinstallieren.

Docker Machine wird auf der Webseite von GitHub zum Download angeboten. Hier der Link zur Download-Seite:

https://github.com/docker/machine/releases

Suchen Sie dort nach der neuesten Relase-Version (z.B. v0.16.2). Sie befindet sich in der Auflistung auf dieser Webseite ganz oben.

### 19.4.1 Installation von docker-machine unter Windows 10

Am Ende des Eintrags zum jeweiligen Release befindet sich eine Liste der Assets, also der für den Download verfügbaren Dateien (Abb. 19.18):


**Abb. 19.18** Liste der Assets von `docker-machine`

Klicken Sie hier auf den gewünschten Eintrag (z.B. **docker-machine-Windows-x86_64.exe**), um den Download der Datei zu starten.

Nach erfolgreichem Download befindet sich die exe-Datei (**docker-machine-Windows-x86_64.exe**) in dem Ordner, den Sie als Ziel für den Download der Datei angegeben haben.

Kopieren oder verschieben Sie diese Datei in den Ordner mit den ausführbaren Docker-Dateien. Falls Sie bei der Installation von Docker Desktop nichts anderes angegeben haben, ist das normalerweise:

```
1 'C:\Program Files\Docker\Docker\resources\bin'
```

Benennen Sie zuletzt diese Datei dort um in

```
1 'docker-machine.exe'.
```

Sowohl beim Kopieren als auch beim Umbennen werden Sie in einem Dialogfenster nach Administratorberechtigungen gefragt. Klicken Sie hier auf die Schaltfläche [Fortsetzen] (Abb. 19.19):

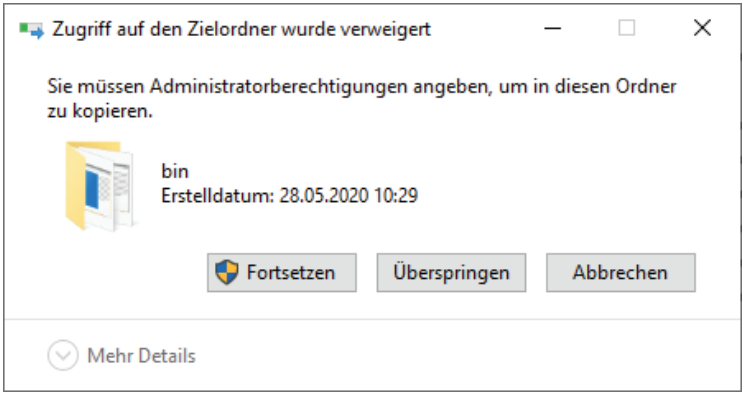



**Abb. 19.19** Bestätigungsdialog für Administrator-Berechtigungen

Um die Installation zu testen, starten Sie eine Shell, z.B. die PowerShell, und geben Sie dort das folgende Kommando ein:

```
1 > docker-machine -version
```

Wenn die Installation erfolgreich war, erhalten Sie von diesem Kommando die Nummer der installierten Version von `docker-machine` (Abb. 19.20):

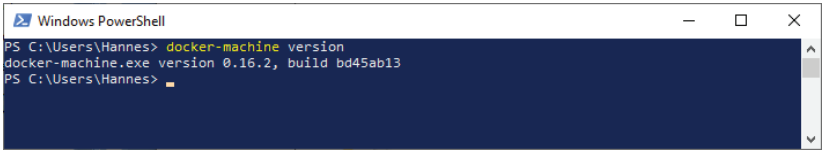



**Abb. 19.20** Test der Installation von `docker-machine`

### 19.4.2 Installation von docker-machine unter Linux

Zur Installation von Docker Machine unter Linux wird auf der GitHub-Webseite ein `curl`-Kommando angeboten.

Das Kommando kann hier kopiert, in einer Shell eingefügt und dann ausgeführt werden:

```
$ curl -L \
https://github.com/docker/machine/releases/download/v0.16.2/
docker-machine-\
'uname -s'-'uname -m' >/tmp/docker-machine && \
    chmod +x /tmp/docker-machine && \
    sudo cp /tmp/docker-machine /usr/local/bin/docker-machine
```

### 19.4.3 Installation von docker-machine unter MAC-OS

Auch für die Installation von Docker Machine unter MAC-OS wird auf der GitHub-Webseite ein `curl`–Kommando angeboten.

Um es auszuführen, öffnen Sie ein Terminal-Fenster. Klicken Sie dazu im Dock auf *Programme*, wählen Sie dann *Dienstprogramme* aus und klicken Sie zum Schluss auf die Verknüpfung zum *Terminal*.

Das `curl`-Kommando kann ebenfalls kopiert, im Terminal eingefügt und dann ausgeführt werden:

```
$ curl \
-L https://github.com/docker/machine/releases/download/v0.16.2/
docker-machine-\
'uname -s'-'uname -m' >/usr/local/bin/docker-machine && \
  chmod +x /usr/local/bin/docker-machine
```

## 19.5 Virtuellen Computer mit UBUNTU erstellen

Der Hyper-V Manager bietet unter anderem die Möglichkeit, auf sehr einfache Art und Weise virtuelle System zu erstellen.

Um den Hyper-V Manager zu starten, geben Sie im Suchfeld der Windows-Taskleiste die Zeichenfolge „Hyper-V“ ein. Im Ergebnisfenster klicken Sie dann auf das Hyper-V Icon.

Nach dem Start sehen Sie das Hauptfenster des Hyper-V Managers (Abb. 19.21):

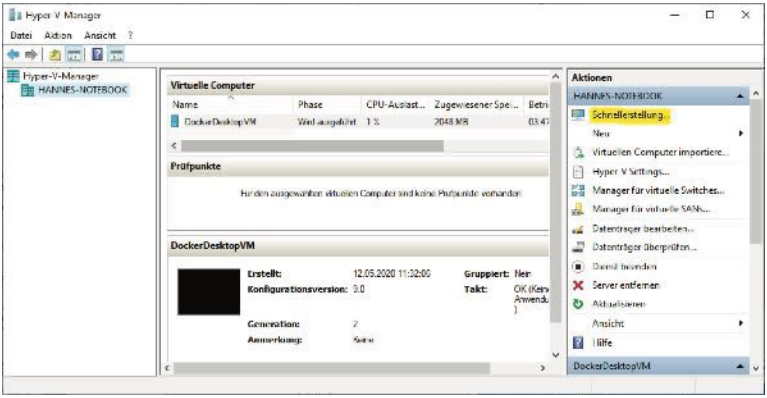



**Abb. 19.21** Hauptfenster des Hyper-V Managers

Im rechten Bereich des Programmfensters befindet sich die Gruppe *Aktionen*. Wählen Sie dort den Eintrag SCHNELLERSTELLUNG... aus.

Damit öffnet sich das Fenster „Virtuellen Computer erstellen“ (Abb. 19.22).

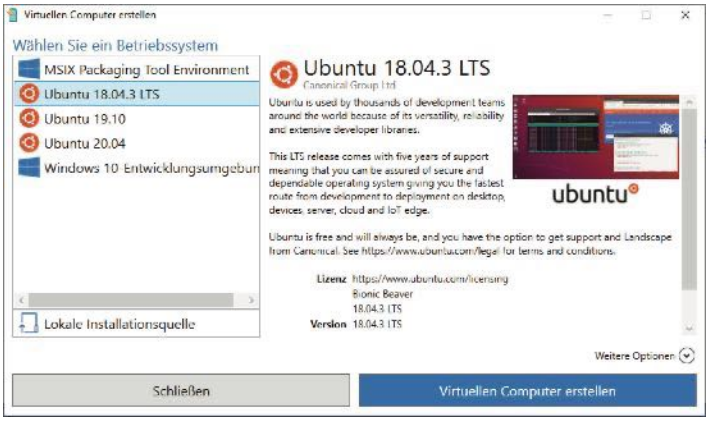



**Abb. 19.22** Fenster „VIRTUELLEN COMPUTER ERSTELLEN“ des Hyper-V Managers

Wählen Sie hier ein Betriebssystem aus (hier zum Beispiel *Ubuntu 10.04.3 LTS*). Und klicken dann mit der Maus auf die Schaltfläche [Virtuellen Computer erstellen], um die Erstellung eines virtuellen Computers mit dem ausgewählten Betriebssystem zu starten.

Da erscheint zunächst ein Fenster mit einem Progressbar, mit dem anzeigt wird, wieweit der Download des Images fortgeschritten ist.

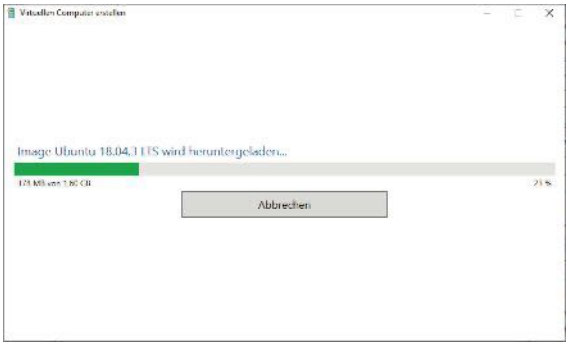



**Abb. 19.23** Fenster „Virtuellen Computer erstellen“: Image wird heruntergeladen.

Am Ende erscheint ein Dialog mit der Meldung „Der virtuelle Computer wurde erfolgreich erstellt“ (Abb. 19.24):


**Abb. 19.24** Fenster „Virtuellen Computer erstellen“: Der virtuelle Computer wurde erfolgreich erstellt.

Klicken Sie jetzt auf die Schaltfläche [Verbinden]. Damit erscheint das Fenster der virtuellen Maschine, die im Moment noch ausgeschaltet ist.

Mit dem Button [Starten] wird die virtuelle Maschine eingeschaltet und Ubuntu startet nun in diesem virtuellen Computer:

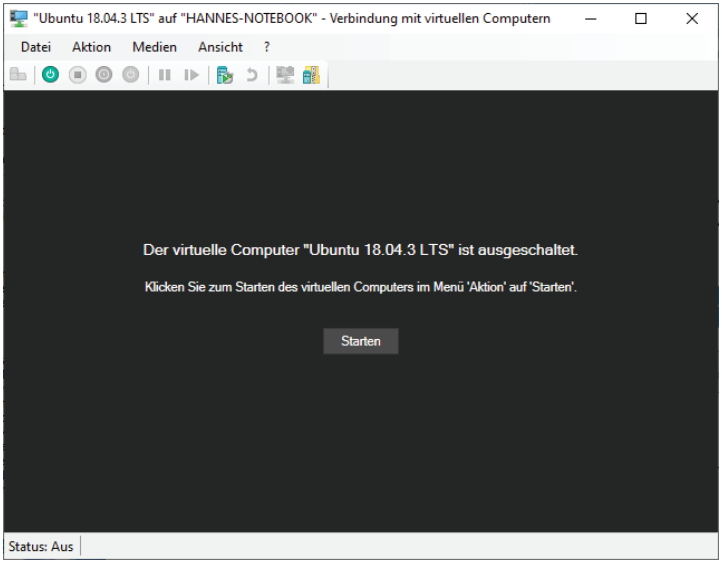



**Abb. 19.25** Fenster „Virtuellen Computer erstellen“: den virtuellen Computer starten

Es werden die Eingaben zu Systemeinstellungen wie Sprache, Tastaturbelegung, Zeitzone und Benutzerdaten abgefragt. Danach wird der Installationsvorgang fortgesetzt, so wie die Installation auf einem physikalischen Rechner auch. Am Ende kann für den virtuellen Computer noch die Größe des Bildschirms eingestellt werden.

Jetzt können Sie auf diesem virtuellen System Docker installieren. Die Anleitung dazu finden Sie ebenfalls im Anhang dieses Buches im Kapitel 19.2.

## 19.6 Das Projekt „Play with Docker“

Play with Docker, das ist eine Web-Applikation, die als „Übungsgelände“ zum Spielen und zum Testen von Docker-Kommandos genutzt werden kann.

Es wird der Zugriff auf virtuelle Maschinen mit dem Betriebssystem „Alpine Linux“ über eine Webseite simuliert. Man kann mit dieser Simulation Docker Container erstellen und starten und eben auch Docker-Cluster im Swarm Mode aufbauen und ausprobieren.

Der einzige Nachteil bei der Arbeit mit dieser Web-Applikation ist, dass eine Sitzung nach vier Stunden Laufzeit wieder beendet und eine während der Sitzung aufgebaute Testumgebung dabei wieder gelöscht wird.

Da das Tool aber eine kostenlose Möglichkeit bietet, um den Umgang mit Docker und dem Swarm Mode mit wenig Aufwand zu üben, ist das aber meiner Ansicht nach ganz in Ordnung.

Zugang zu dieser Web-Applikation erhalten Sie über die folgende URL:

https://labs.play-with-docker.com/

Es erscheint dann die Anmeldeseite von Play with docker (Abb. 19.26):


**Abb. 19.26** Die Anmeldeseite von „Play with Docker“

Klicken Sie hier auf den Button [Login] und dann auf den Eintrag [Docker]. Damit melden Sie sich mit Benutzernamen und Passwort aus Ihrem Docker Account an Play with Docker an. War das erfolgreich, dann wird der Login Button der Anmeldeseite in einen grünen [Start] Button verwandelt. Durch einen Mausklick auf Start wird die Web-Applikation mit einer neuen Session gestartet (Abb. 19.27):

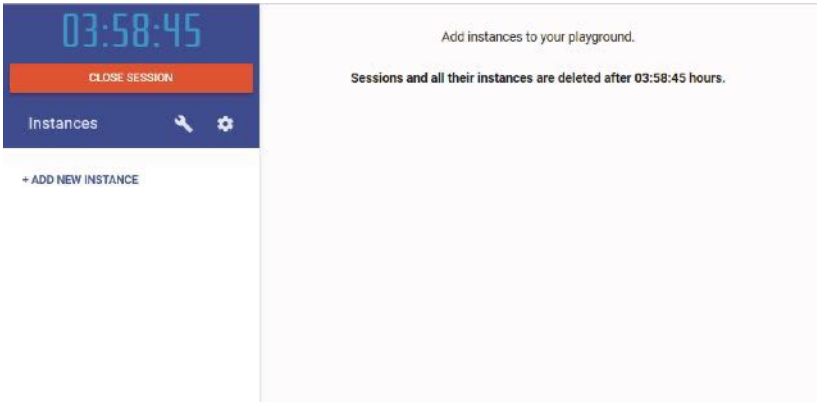



**Abb. 19.27** Beginn einer Session von Play with Docker

Das Fenster ist zu Beginn noch sehr übersichtlich. Links sehen Sie eine Zeitanzeige, welche die abgelaufene Zeit der Session ausgibt. Durch Klick auf den Eintrag [+ ADD NEW INSTANCE] können für die laufende Session neue virtuelle Maschinen erzeugt werden.

Wie erzeugen für diese Einführung drei virtuelle Maschinen:

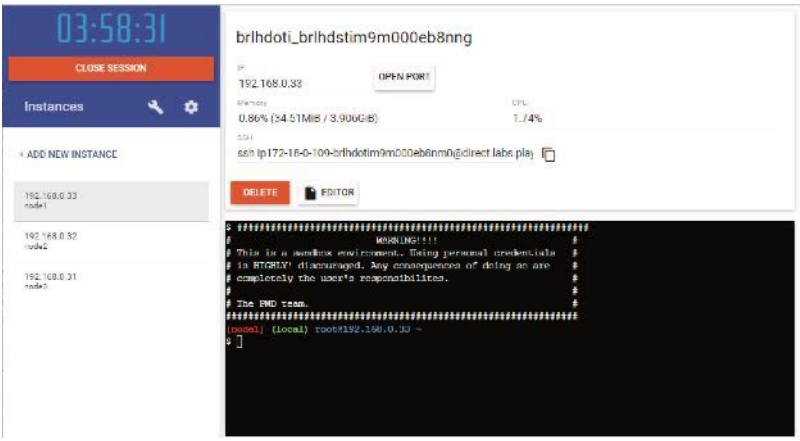



**Abb. 19.28** Play with Docker mit drei virtuellen Nodes

In der linken Spalte steht eine Liste mit den Namen der virtuellen Maschinen (`'node1'`, `node2`, `'node3'`, ...) und deren IP-Adresse.

Wird ein Eintrag dieser Liste mit der Maus markiert, dann werden auf der rechten Seite die Informationen zu diesem Node ausgegeben und zusätzlich im unteren Bereich ein Fenster mit einer Kommando-Shell für diesen Node zur Verfügung gestellt. Hier können Sie jetzt beliebige Linux und eben auch Docker-Kommandos eingeben und ausprobieren.

Hier ein paar Beispiele dazu:

1. Initialisierung des Swarm Modes auf `'node1'`:

   - Aktivieren Sie das Shell-Kommando-Fenster für `'node1'` durch Anklicken des zugehörigen Listeneintrags auf der linken Seite des Fensters mit der Maus.
   - Geben Sie im Shell-Fenster das Kommando ein, um den Swarm Mode für diesen Node zu aktivieren:

```
1 $ docker swarm init --advertise-addr <MANAGER_IP>
```

- Übernehmen Sie als Advertise-Adresse die IP, die im linken Teil des Fensters für `'node1'` angezeigt wird (Abb. 19.29):

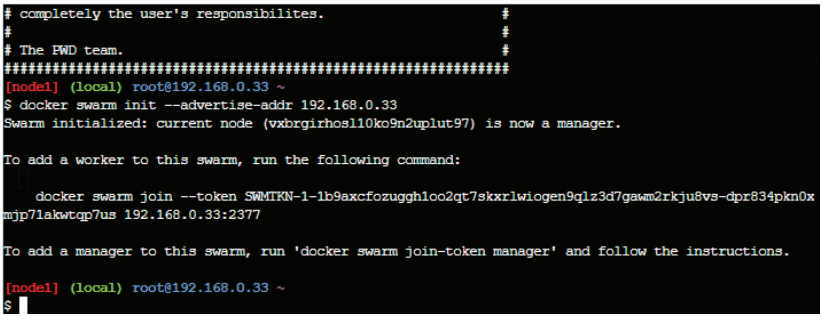



**Abb. 19.29** Play with Docker: Initialisierung des Swarm Modes auf `'node1'`

2. `'node2'` und `'node3'` dem Swarm als Worker Nodes zufügen:

   - Kopieren Sie das `docker swarm join`-Kommando aus dem Shell-Kommando-Fenster von `'node1'` wie folgt in die Zwischenablage. Markieren Sie die Zeichenkette und öffnen Sie ein Kontextmenü durch einen Rechtsklick mit der Maus auf den markierten Text. Wählen Sie aus dem Kontextmenü den Befehl KOPIEREN.
   - Aktiveren Sie das Shell-Kommando-Fenster für `'node2'`.
   - Fügen Sie die Kopie des Kommandos aus der Zwischenablage in die Kommandozeile von `'node2'` wie folgt ein: Zeigen Sie mit der Maus in das Shell-Kommando-Fenster von `'node2'` und öffnen Sie ein Kontextmenü durch einen Rechtsklick mit der Maus im Fenster. Wählen Sie aus dem Kontextmenü den Befehl KOPIEREN.
   - Starten Sie das Kommando im Shell-Kommando-Fenster mit der Eingabetaste (Abb. 19.30):

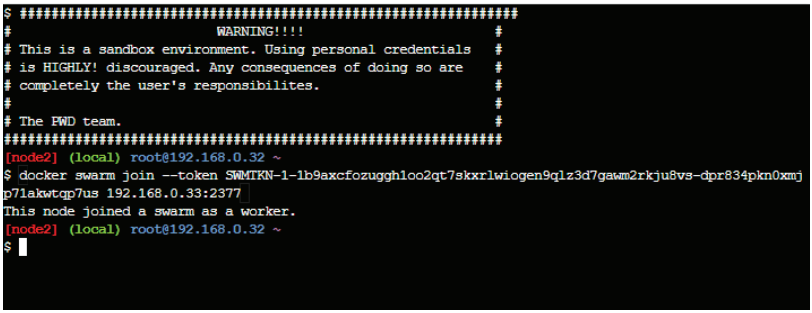



**Abb. 19.30** Play with Docker: `'node2'` dem Swarm als Worker Nodes zufügen

- Wiederholen Sie die obigen Aktionen für `'node3'`.

3. Informationen des Swarms anzeigen lassen:

- Aktivieren Sie wieder das Shell-Kommando-Fenster für `'node1'`.
- Lassen Sie sich eine Liste der Nodes für den Swarm ausgeben:

```
1  docker node ls
```

- Lassen Sie sich ausführliche Informationen über `'node1'` ausgeben (Abb. 19.31):

```
1  docker info
```

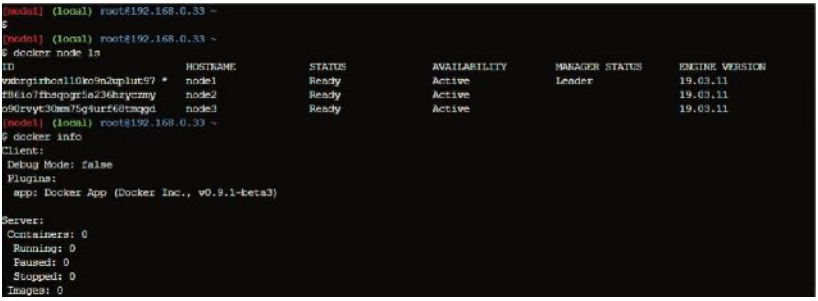



**Abb. 19.31** Play with Docker: Informationen des Swarms anzeigen lassen

4. Node zum Manager befördern:

   - Aktivieren Sie, wenn nötig, das Shell-Kommando-Fenster für `'node1'`.
   - Geben Sie das Kommando ein, um `'node2'` zum Manager Node zu „befördern“und lassen Sie sich die Änderung in der Node-Liste ausgeben (Abb. 19.32):

```
$ docker node update --role manager node2
$ docker node ls
```

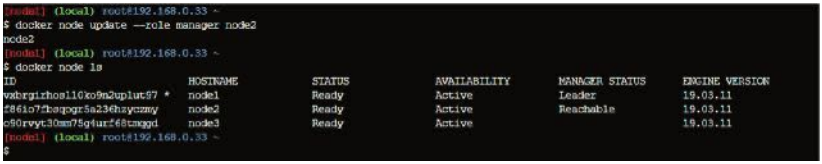



**Abb. 19.32** Play with Docker: `'node2'` zum Manager hochstufen

5. Node zum Worker herabstufen:

   - Geben Sie das Kommando ein, um `'node2'` vom Manager Node zum Worker Node herabzustufen und lassen Sie sich die Änderung in der Node-Liste ausgeben (Abb. 19.33):

```
$ docker node update --role worker node2
$ docker node ls
```


**Abb. 19.33** Play with Docker: `'node2'` zum Worker herabstufen

6. Node entfernen:

   - Geben Sie das Kommando ein, um `'node2'` zu entfernen:

```
$ docker node rm
```

   - Es wird eine Fehlermeldung ausgegeben. Ein aktiver Node kann nicht entfernt werden. Wir müssen `'node2'` erst beenden. Bei Play with Docker erledigen wir das, indem wir `'node2'` markieren und im rechten Bereich des Fensters den Button [DELETE] klicken. Anschließend können wir aus dem Shell-Kommando-Fenster von `'node1'` heraus den `'node2'` erfolgreich entfernen. Wir lassen uns das zur Kontrolle auch noch mit dem ls-Kommando ausgeben (Abb. 19.34):

```
$ docker node rm
$ docker node ls
```

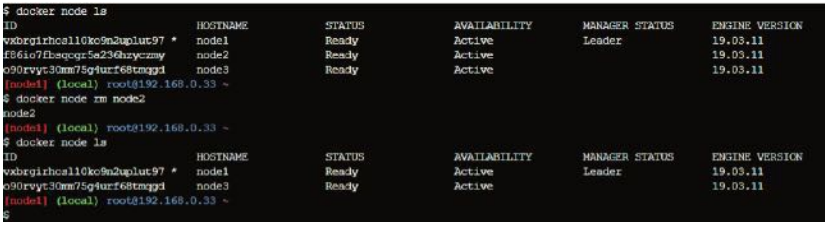



**Abb. 19.34** Play with Docker: `'node2'` entfernen

An dieser Stelle beenden wir die Einführung in die Web-Applikation „Play with Docker". Jetzt sind Sie an der Reihe und können, wenn Sie wollen, weitere Docker-Aktionen unter „Play with Docker" ausprobieren und üben.

Viel Spaß dabei!

## 19.7 Das Projekt „Play with Kubernetes"

So wie es für Docker das Projekt „Play with Docker" gibt, wird jetzt für Kubernetes ebenfalls eine Web-Applikation angeboten. Diese erlaubt es

Kubernetes zu testen bzw. den Umgang mit `kubectl`-Kommandos zu üben. Sie hat den Namen „Play with Kubernetes“.

Auch hier werden virtuelle Maschinen mit dem Betriebssystem „Alpine Linux“ über die Webseite bereitgestellt und wie schon bei „Play with Docker“ endet eine Sitzung nach vier Stunden Laufzeit und die Testumgebung ist danach wieder verschwunden.

Der Zugang zu „Play with Kubernetes“ erfolgt über folgende URL:

https://labs.play-with-k8s.com/

Es erscheint dort die Login-Seite von „Play with Kubernetes“.

Klicken Sie hier auf den Button [Login] und dann auf den Eintrag [Docker]. Damit melden Sie sich mit Benutzernamen und Passwort aus Ihrem Docker Account bei „Play with Kubernetes“ an. War das erfolgreich, dann wird der Login Button der Anmeldeseite in einen grünen [Start] Button geändert (Abb. 19.35).


**Abb. 19.35** Die Startseite von „Play with Kubernetes“

Durch einen Mausklick auf Start wird die Web-Applikation mit einer neuen Session gestartet (Abb. 19.36):

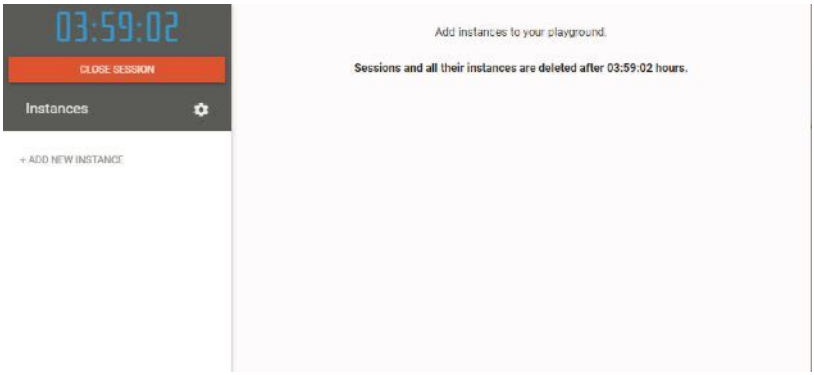



**Abb. 19.36** Beginn einer Session von „Play with Kubernetes“

Die Gemeinsamkeiten zu Projekt „Play with Docker“ sind nicht zu übersehen. Wenn Sie „Play with Docker“ bereits ausprobiert haben, dann werden Sie sich sicher auch hier schnell zurechtfinden.

Das Fenster ist zu Beginn noch komplett identisch zu dem des Docker-Projekts. Links befindet sich die Zeitanzeige mit der abgelaufene Session-Zeit. Durch Klick auf den Eintrag [+ ADD NEW INSTANCE] werden für die laufende Session neue virtuelle Maschinen erzeugt.

Erzeugen wir zum Testen drei virtuelle Maschinen (Abb. 19.37):

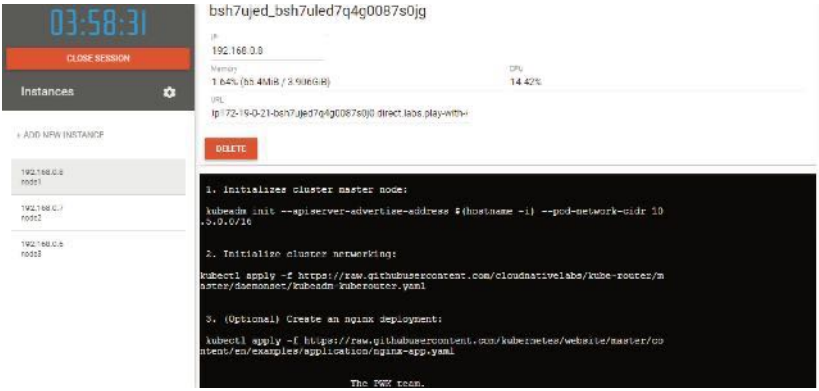



**Abb. 19.37** „Play with Kubernetes“ mit drei virtuellen Nodes

Links befindet sich die Liste mit den Namen der virtuellen Maschinen (`'node1'`, `'node2'`, `'node3'`) und deren IP-Adresse.

Wird ein Eintrag dieser Liste mit der Maus markiert, dann werden auf der rechten Seite die Informationen zu diesem Node ausgegeben und zusätzlich im unteren Bereich ein Fenster mit einer Kommando-Shell für diesen Node zur Verfügung gestellt. Hier können Sie jetzt beliebige Linux-, Docker- oder Kubernetes-Kommandos eingeben und ausprobieren.

Beim Start eines Kommando-Fensters für einen Node erscheinen zuerst Informationen zur Initialisierung eines Master Nodes für ein neues Cluster. Die dort angezeigten Kommandos kann man per Copy und Paste an den Eingabeprompt des Fensters übertragen und ausführen (unter Umständen müssen Sie dafür mit der rechten Maustaste das Kontext-Menü öffnen und dort die Menüpunkte KOPIEREN und EINFÜGEN auswählen).

Beginnen wir mit der Initialisierung des Master Nodes für das Cluster:

- Aktivieren Sie das Shell-Kommando-Fenster für `'node1'` durch Anklicken des zugehörigen Listeneintrags auf der linken Seite des Fensters mit der Maus.
- Kopieren Sie im Shell-Fenster das Kommando kubeadm `init` an den Eingabeprompt des Kommandofensters von `'node1'`, um den Master Mode für diesen Node zu initialisieren:

```
1 $ kubeadm init --apiserver-advertise-address $(hostname -i)
2 --pod-network-cidr 10.5.0.0/16
```

- Führen Sie das Kommando aus und warten Sie ab. Die Initialisierung dauert eine Weile. Es werden eine ganze Reihe von Aktionen durchgeführt. Diese sind nötig, um ein neues Cluster zu initialisieren und ein API-Interface zu für die Kommunikation zu konfigurieren (Abb. 19.38).

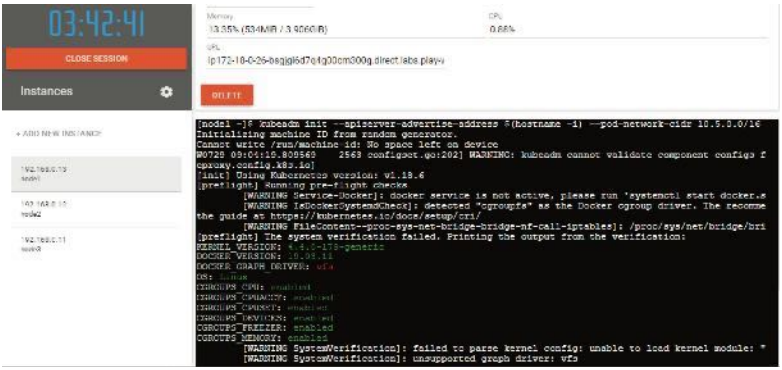



**Abb. 19.38** Play with Kubernetes: Initialisierung des Master Modes auf `'node1'`

In der Ausgabe des `kubeadm`-Kommandos erscheint eine Liste mit Kommandos, die zum Start eines Clusters normalerweise notwendig wären:

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

„Play with Kubernetes“ hat das aber schon für uns erledigt und wir können diese Meldung einfach ignorieren.

Sehen wir uns jetzt einmal die Nodes des neuen Clusters an. Geben Sie dazu das Kommando `kubectl get nodes` ein (Abb. 19.39):

```
[node1 ~]$ kubectl get nodes
NAME     STATUS     ROLES    AGE     VERSION
node1    NotReady   master   9m25s   v1.18.4
[node1 ~]$
```

**Abb. 19.39** Play with Kubernetes: Anzeige der Nodes ohne Pod-Netzwerk

Es gibt im Moment nur den Master Node im Cluster und der hat den Status `NotReady`. Das liegt daran, dass noch kein Pod-Netzwerk eingerichtet ist.

Im nächsten Schritt initialisieren wir deshalb das Pod-Netzwerk (Abb. 19.40):

- Das Kommando dazu wurde uns auch schon von PWK mit dem zweiten Hinweis beim Start des Nodes angezeigt. Es ist ein `kubectl apply`-Kommando:

```
$ kubectl apply -f https://raw.githubusercontent.com/
cloudnativelabs/kube-router/master/daemonset/kubeadm-kuberouter.
yaml
```

- Das kann man wieder per Copy und Paste an den Eingabeprompt des Shell-Fensters übertragen und ausführen.
- Anschließend warten wir noch kurz und überprüfen dann noch einmal den Cluster-Status. Der sollte sich nach einer gewissen Zeit auf Ready ändern.

```
$ kubectl get nodes
```

```
[node1 ~]$ kubectl apply -f https://raw.githubusercontent.com/cloudnativelabs/kube-router/master/daemonset/kubeadm-kuberouter.yaml
configmap/kube-router-cfg created
daemonset.apps/kube-router created
serviceaccount/kube-router created
clusterrole.rbac.authorization.k8s.io/kube-router created
clusterrolebinding.rbac.authorization.k8s.io/kube-router created
[node1 ~]$ kubectl get pods
No resources found in default namespace.
[node1 ~]$ kubectl get nodes
NAME    STATUS   ROLES    AGE    VERSION
node1   Ready    master   103s   v1.18.4
[node1 ~]$
```

**Abb. 19.40** Play with Kubernetes: Initialisierung des Pod-Netzwerks

Nachdem das Pod-Netzwerk fertig initialisiert ist, können wir das Cluster erweitern, indem wir `'node2'` und `'node3'` als Worker Nodes zufügen.

- Das benötigte `kubeadm join`-Kommando wurde am Ende der Ausführung des `kubeadm init`-Kommandos als Teil der Ausgabe angezeigt (Abb. 19.41):

```
Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.0.33:6443 --token js6fvb.lcphmkkd6ie5f89h \
    --discovery-token-ca-cert-hash sha256:fb4991579575dbed26234d0e3e2fb70ea8aa89449363ebd7d88c3e0141077632
```

**Abb. 19.41** Play with Kubernetes: Ausgabe des kubeadm join-Kommando von `kubeadm init`

- Kopieren Sie das `kubeadm join`-Kommando aus dem Shell-Kommando-Fenster von `'node1'` in die Zwischenablage.
- Aktiveren Sie das Shell-Kommando-Fenster für `'node2'`.
- Fügen Sie die Kopie des Kommandos aus der Zwischenablage am Eingabeprompt von `'node2'` ein.
- Starten Sie das Kommando im Shell-Kommando-Fenster mit der Eingabetaste (Abb. 19.42):

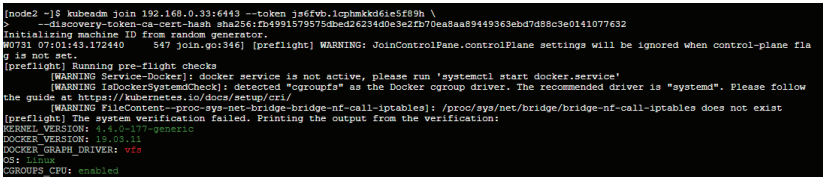



**Abb. 19.42** Play with Kubernetes: node2 dem Kubernetes Cluster als Worker Node zufügen

- Wiederholen Sie die obigen Aktionen für `'node3'`.

Testen des Clusters:

- Wechseln Sie auf `'node1'`, den Manager Node. Lassen Sie sich dort die aktuellen Node-Informationen anzeigen:

```
1  $ kubectl get nodes
```

```
[node1 ~]$
[node1 ~]$ kubectl get nodes
NAME    STATUS   ROLES    AGE   VERSION
node1   Ready    master   43m   v1.18.4
node2   Ready    <none>   39m   v1.18.4
node3   Ready    <none>   37m   v1.18.4
[node1 ~]$
```

**Abb. 19.43** Play with Kubernetes: ein Cluster mit drei Nodes

- Prüfen Sie die Versionen von Docker und `kubectl`:

```
1  $ docker version
2  $ kubectl version
```

Test Deployment von NGINX erstellen:

- Bei der Erstellung der Node-Instanzen wurde von PWK als dritter Hinweis das `kubectl apply`-Kommando angezeigt, mit dem ein Deployment aus dem NGINX Image erstellt werden kann:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/
website/master/content/en/examples/application/nginx-app.yaml
```

- Kopieren Sie das Kommando an den Eingabeprompt von `'node1'` und führen es aus (Abb. 19.44):

```
[node1 ~]$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/application/nginx-app.yaml
service/my-nginx-svc created
deployment.apps/my-nginx created
```

**Abb. 19.44** Play with Kubernetes: erstellen eines NGINX-Test-Deployments

- Jetzt können Sie das Deployment überprüfen (Abb. 19.45):

```
$ kubectl get deployment
$ kubectl get pods
```

```
[node1 ~]$
[node1 ~]$ kubectl get deployment
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
my-nginx   3/3     3            3           2m25s
[node1 ~]$
[node1 ~]$ kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
my-nginx-6b474476c4-9sdsp   1/1     Running   0          2m34s
my-nginx-6b474476c4-nnbxl   1/1     Running   0          2m34s
my-nginx-6b474476c4-vb7q7   1/1     Running   0          2m34s
[node1 ~]$
```

**Abb. 19.45** Play with Kubernetes: Überprüfung des NGINX-Test-Deployments

- Wir lassen uns noch die Details des Deployments anzeigen (Abb. 19.46):

```
$ kubectl describe deployment my-nginx
```

```
[node1 ~]$ kubectl describe deployment  my-nginx
Name:                   my-nginx
Namespace:              default
CreationTimestamp:      Fri, 31 Jul 2020 07:13:28 +0000
Labels:                 app=nginx
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=nginx
Replicas:               3 desired | 3 updated | 3 total | 2 available | 1 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx:
    Image:        nginx:1.14.2
    Port:         80/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
```

**Abb. 19.46** Play with Kubernetes: Details des NGINX-Test-Deployments

- Sehen wir uns auch die Details des Service an. Der Name wurde während der Erstellung der Anwendung als Ausgabe des Kommandos `kubectl apply` bekanntgegeben (Abb. 19.47):

```
1  $ kubectl describe svc my-nginx-svc
```

```
[node1 ~]$
[node1 ~]$ kubectl describe svc my-nginx-svc
Name:                     my-nginx-svc
Namespace:                default
Labels:                   app=nginx
Annotations:              Selector:  app=nginx
Type:                     LoadBalancer
IP:                       10.108.165.93
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  30674/TCP
Endpoints:                10.5.1.2:80,10.5.2.2:80,10.5.2.3:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

**Abb. 19.47** Play with Kubernetes: Details des Service `my-nginx-svc` der NGINX-Test-Anwendung

Soviel zur Einführung in „Play with Kubernetes“. Jetzt sind wieder Sie an der Reihe. Probieren Sie weitere Kubernetes-Aktionen auf der Spielwiese von „Play with Kubernetes“ aus.

Viel Spaß beim Üben!

## 19.8 Ein Minikube-Cluster für Docker unter Ubuntu Linux anlegen

Wenn Sie unter Linux arbeiten und dort zum Testen oder zum Lernen von Kubernetes eine Lösung für ein Single Node-Cluster einrichten wollen, dann stellt das Tool ***Minikube*** dafür eine einfache Lösung bereit.

Minikube ist ein Open Source Tool, mit dem es möglich wird, Kubernetes lokal auf einem System auszuführen. Es wird ein Single Node-Cluster innerhalb einer virtuellen Maschine ausgeführt, die man sich ähnlich wie eine virtuelle Maschine von Docker Machine vorstellen kann.

In dieser virtuellen Maschine ist übrigens auch Docker bereits vorinstalliert.

Minikube könnte auch zusammen mit Windows oder Mac Systemen genutzt werden. Da dort aber Single Node-Cluster mithilfe von Docker Desktop inzwischen deutlich komfortabler realisiert werden können, gehen wir in diesem Buch auf die Einrichtung von Minikube für diese Systeme nicht weiter ein.

### 19.8.1 Installation von Minikube auf Ubuntu Linux

Um Minikube auf einem Ubuntu Linux System zu installieren, müssen die nachfolgend beschriebenen Schritte ausgeführt werden.

Im ersten Schritt aktualisieren wir den APT Package Index für unsere Ubuntu-Distribution:

```
1 $ sudo apt-get update
```



Dann installieren wir Pakete, die notwendig sind, um Repositories über HTTPS zu nutzen:

```
1 $ sudo apt-get install apt-transport-https
```

Sicherheitshalber wird noch ein Upgrade durchgeführt, damit alle Pakete des Systems aktuell sind:

```
$ sudo apt-get upgrade
```

Damit man Minikube nutzen kann, ist es nötig, dass `kubectl` installiert ist. Das folgende `curl`-Kommando führt die Installation von `kubectl` aus, danach wird für die Datei das Execute Flag gesetzt und die Datei in das Verzeichnis `/usr/local/bin` verschoben:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/
release/$(curl -s https://storage.googleapis.com/kubernetes-
release/ release/stable.txt)/bin/linux/amd64/kubectl
$ chmod +x ./kubectl
$ sudo mv ./kubectl /usr/local/bin/kubectl
```

Alternativ kann bei Ubuntu auch das `apt-get`-Kommando für die Installation genutzt werden:

```
$ sudo apt-get install -y kubectl
```

Nach der Installation von `kubectl` prüfen wir, ob die diese erfolgreich war. Dazu fragen wir die installierte Version ab:

```
$ kubectl version
```

Minikube benötigt weiterhin einen Treiber für die Virtualisierung. Zu diesem Zweck wird die Installation von VirtualBox durchgeführt:

```
$ sudo apt install virtualbox virtualbox-ext-pack
```

Jetzt sind alle Vorbereitungen getroffen, damit Minikube installiert werden kann:

```
$ curl -Lo minikube https://storage.googleapis.com/minikube/
releases/latest/minikube-linux-amd64 && chmod +x minikube
$ sudo mv ./minikube /usr/local/bin/minikube
```

Auch hier fragen wir die Version von Minikube ab, um die Installation zu prüfen:

```
$ minikube version
```

### 19.8.2 Minikube anwenden

War die Installation erfolgreich, dann können wir schon loslegen und Minikube starten:

```
$ minikube start
```

Hier noch das `minikube start`-Kommando mit der Angabe des Treibers für die virtuelle Maschine als Parameter:

```
$ minikube start --driver=virtualbox
```

Mit dem Start von Minikube haben wir jetzt unser lokales Single Node-Cluster und können mit der Eingabe von `kubectl`-Kommandos beginnen.

Als Erstes lassen wir uns die Cluster-Informationen ausgeben:

```
$ kubectl cluster-info
```

Der Screenshot zeigt zuerst die Ausgabe bei der Abfrage der Version von Minikube, dann die Ausgaben beim Start von Minikube und zuletzt das Ergebnis aus der Cluster-Info-Abfrage (Abb. 19.48):

```
$ minikube version
minikube version: v1.8.1
commit: cbda04cf6bbe65e987ae52bb393c10099ab62014
$ minikube start
* minikube v1.8.1 on Ubuntu 18.04
* Using the none driver based on user configuration
* Running on localhost (CPUs=2, Memory=2460MB, Disk=145651MB) ...
* OS release is Ubuntu 18.04.4 LTS
* Preparing Kubernetes v1.17.3 on Docker 19.03.6 ...
  - kubelet.resolv-conf=/run/systemd/resolve/resolv.conf
* Launching Kubernetes ...
* Enabling addons: default-storageclass, storage-provisioner
* Configuring local host environment ...
* Waiting for cluster to come online ...
* Done! kubectl is now configured to use "minikube"
$ kubectl cluster-info
Kubernetes master is running at https://172.17.0.22:8443
KubeDNS is running at https://172.17.0.22:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

**Abb. 19.48** Die ersten Minikube-Befehle unter Ubuntu Linux

Hier noch die zwei `kubectl`-Kommandos, um Node- und Pod-Informationen abzufragen, und ein Screenshot mit den Ergebnissen der Ausführung (Abb. 19.49):

```
$ kubectl get nodes --all-namespaces
$ kubectl get pods --all-namespaces
```

```
$ kubectl get nodes --all-namespaces
NAME       STATUS   ROLES    AGE   VERSION
minikube   Ready    master   44s   v1.17.3
$ kubectl get pods --all-namespaces
NAMESPACE     NAME                               READY   STATUS    RESTARTS   AGE
kube-system   coredns-6955765f44-5s5xg           1/1     Running   0          44s
kube-system   coredns-6955765f44-hhh29           1/1     Running   0          44s
kube-system   etcd-minikube                      1/1     Running   0          48s
kube-system   kube-apiserver-minikube            1/1     Running   0          48s
kube-system   kube-controller-manager-minikube   1/1     Running   0          48s
kube-system   kube-proxy-5dnw4                   1/1     Running   0          44s
kube-system   kube-scheduler-minikube            1/1     Running   0          48s
kube-system   storage-provisioner                1/1     Running   0          48s
$
```

**Abb. 19.49** `kubectl`-Befehle mit Minikube

Die folgende Tabelle enthält eine Auswahl der gebräuchlichsten CLI-Kommandos von Minikube:

| | |
|---|---|
| minikube help | Zeigt die Minkube-Onlinehilfe an |
| minikube start | Startet ein lokalen Kubernetes-Cluster |
| minikube stop | Stoppt ein laufendes Kubernetes-Cluster |
| minikube delete | Löscht ein lokales Kubernetes-Cluster |
| minikube status | Zeigt den Status des lokalen Kubernetes-Cluster an. |

### 19.8.3 Online Installationen von Minikube Terminals

Wenn Sie den Umgang mit Minikube mit möglichst wenig Aufwand probieren möchten, dann können Sie im Internet auf Online-Terminals zugreifen auf denen Minikube installiert ist. Diese Funktion wird Ihnen unter anderem auf den Webseiten von *Katacoda* angeboten:

https://katacoda.com/courses/kubernetes/launch-single-node-cluster

Die Internet Seiten von Kubernetes stellen ebenfalls solche Online-Terminals bereit:

https://kubernetes.io/docs/tutorials/hello-minikube/

## 19.9 Übersicht der Dockerfile-Anweisungen

Hier zuerst das allgemeine Format von Anweisungen in Dockerfiles:

Kommentarzeilen beginnen mit dem Hash-Zeichen.

```
1  # Kommentar
```

Anweisungen in Dockerfiles sind nicht case-sensitive, werden aber üblicherweise in Großbuchstaben angegeben, um sie leichter von den Argumenten unterscheiden zu können.

```
1  ANWEISUNG argument
```

Die erste Anweisung in einem Dockerfile muss normalerweise die FROM-Anweisung sein, sie kann aber nach Kommentaren der ARG-Anweisung oder Parser-Direktiven stehen.

| Anweisung | Beschreibung |
|---|---|
| FROM <image> | Mit dieser Anweisung wird das Parent Image angegeben.<br>Beispiel:<br>FROM ubuntu:18.4 |
| ARG | Mit der ARG-Anweisung werden Argumente definiert, die an anderer Stelle im Dockerfile benutzt werden können. |
| RUN <command> | Mit dem RUN-Kommando werden beliebige Anweisungen im neuen Image ausgeführt.<br>Beispiel:<br>RUN chmod +x ./hello.sh |

| | |
|---|---|
| CMD [„executable“,“param1“,“param2“] | Es darf nur eine CMD-Anweisung in einem Dockerfile geben. Sind dort mehrere RUN-Einträge vorhanden, wird die letzte ausgeführt. Die Anweisung wird nicht ausgeführt, wenn der Container gebaut wird, sondern bei Containerstart. Hier bestimmt man, welche Applikation bei Containerstart ausgeführt wird.<br>Beispiel:<br>CMD [„/bin/bash“] |
| LABEL<key>=<value> ... | Ein LABEL ist ein Key-Value-Paar. Damit ist es möglich, Images mit Metadaten zu kennzeichnen. Einem Image kann mehr als nur ein Label zugewiesen werden.<br>Beispiel:<br>LABEL version=“1.0.1“ |
| ENV <key>=<value> | Mit der ENV-Anweisung werden Umgebungsvariablen für ein Image als Key-Value-Paar definiert.<br>Beispiel:<br>ENV debug=“true“ |
| EXPOSE <port> | Mit der EXPOSE-Anweisung wird dokumentiert, auf welchen Netzwerk-Port ein Container zur Laufzeit hört. Diese Angabe veröffentlicht einen Port nicht automatisch, sondern muss eher als Dokumentation verstanden werden. Tatsächlich veröffentlicht werden die Ports mit dem Parameter –p bei Ausführung des `docker run`-Kommandos.<br>Beispiel:<br>EXPOSE 8080 |

| | |
|---|---|
| ADD <source> ... <dest> | Die ADD-Anweisung kopiert Dateien, Verzeichnisse oder entfernte Datei-URLs von <source> und fügt sie im Dateisystem an das mit <destination> bestimmeten Zielort ein.<br>Beispiel:<br>ADD http://www.mypagw.com/index.html /usr/share/nginx/html |
| COPY <source> ... <dest> | Die COPY-Anweisung funktioniert ähnlich wie die ADD-Anweisung, mit dem Unterschied, dass als Quelle nur Dateien und Verzeichnisse angegeben werden können, die sich lokal auf dem Hostsystem befinden. Die Angabe von URLs entfernter Dateien ist hier nicht möglich.<br>Beispiel:<br>COPY html /usr/share/nginx/html |
| ENTRYPOINT [„executable“,“param1“,“param2“] | Mit der ENTRYPOINT-Anweisung ist es möglich, für ein Image eine Anweisung zu definieren, die beim Start des Containers ausgeführt wird.<br>Beispiel:<br>ENTRYPOINT [„test“, „-b“] |
| VOLUME <vol> | Durch die VOLUME-Anweisung wird ein „Mount Point“ in einem Container erstellt. Über diesen kann ein externes Verzeichnis aus dem Host eingebunden werden.<br>Beispiel:<br>VOLUME /http |

| | |
|---|---|
| USER <user>[:<group>]<br><br>USER <UID>[:<GID>] | Durch die USER-Anweisung wird ein Benutzer mit einem Namen oder einer UID oder alternativ ein Gruppenname oder eine GID angegeben, der/die während der Laufzeit bei Bau eines Images für die Kommandos RUN, CMD oder ENTRYPOINT verwendet wird.<br>Beispiel:<br>USER hannes |
| WORKDIR <workdir> | Die WORKDIR-Anweisung legt das Arbeitsverzeichnis für die Kommandos RUN, CMD, ENTRYPOINT, COPY und ADD fest.<br>Beispiel:<br>WORKDIR /home/$USER/project |
| SHELL <shell> | Die SHELL-Anweisung erlaubt es, die Standard Shell, die für die Ausführung der Kommandos RUN, CMD und ENTRYPOINT verwendet wird, zu überschreiben.<br>Beispiel:<br>SHELL [„cmd“] |

Ausführliche Informationen über alle möglichen Anweisungen für Dockerfiles finden Sie auf den offiziellen Internetseiten von Docker:

https://docs.docker.com/engine/reference/builder/

## 19.10 Übersicht der Docker CLI-Kommandos

Das Basis-Kommando für alle Docker CLI-Anweisungen hat die folgende Syntax:

```
1 docker <COMMAND> [<PARAMETER> […]]
```

| Anweisung | Beschreibung |
|---|---|
| `docker attach` | Bindet Standard Input, Output und Error an einen laufenden Container.<br>Syntax:<br>`docker attach [<OPTIONS>] CONTAINER`<br>Beispiel:<br>`docker attach my_alpine` |
| `docker build` | Erstellt ein Image auf Basis eines Dockerfiles, welches sich im angegebenen Pfad befindet.<br>Syntax:<br>`docker build [<OPTIONS>] <PFAD>`<br>Beispiel:<br>`docker build -t dockertest:1.0.`<br>Erstellt ein Image auf Basis des Dockerfiles aus dem aktuellen Verzeichnis mit dem Namen „dockertest“ und dem Tag 1.0. |
| `docker config create` | Konfiguration aus einer Datei oder der Standard-Eingabe STDIN erstellen. Wird als letzter Parameter ein – Zeichen anstelle eines Dateinamens angegeben, so wird STDIN als Eingabemedium verwendet.<br>Syntax:<br>`docker config create [<OPTIONS>] <CONFIG> <FILE>\|-`<br>Optionen:<br>-l \| --label Config mit Label erstellen.<br>Beispiel:<br>`docker config create ` `--label rev=dev ` `my_config` |

| | |
|---|---|
| `docker config inspect` | Ausgabe von ausführlichen Informationen über ein oder über mehrere Konfigurationen.<br>Syntax:<br>`docker config inspect [<OPTIONS>] <CONFIG> [<CONFIG> ...]`<br>Optionen:<br>-f \| --format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates.<br>--pretty Ausgabe in einem leicht lesbaren Format erzeugen.<br>Beispiel:<br>`docker config inspect --pretty my_config` |
| `docker config ls` | Auflisten aller Konfigurationen in einem Swarm.<br>Syntax:<br>`docker config ls [<OPTIONS>]`<br>Optionen:<br>-- format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates.<br>Beispiel:<br>`docker config ls` |
| `docker config rm` | Entfernen einer oder mehrerer Konfigurationen.<br>Syntax:<br>`docker config rm <SECRET> [<SECRET> ...]`<br>Beispiel:<br>`docker config rm my_config` |

| | |
|---|---|
| `docker con tainer attach` | Verbinden von den Streams STDIN, STDOUT und STDERR mit einem laufenden Container.<br>Syntax:<br>`docker container attach <CONTAINER>`<br>Beispiel:<br>`docker container attach my_alpine` |

| | |
|---|---|
| `docker container inspect` | Zeigt detaillierte Informationen von einem oder mehreren Containern.<br>Syntax:<br>`docker container inspect [<OPTIONS>] <CONTAINER> [<CONTAINER> …]`<br>Optionen:<br>-f, --format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates.<br>Beispiel:<br>`docker container inspect nginx` |
| `docker container ls` | Auflisten aller Container<br>Syntax:<br>`docker container ls [<OPTIONS>]`<br>Optionen:<br>-f, --format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates. |
| `docker container pause` | Pausiert alle Prozesse innerhalb von einem oder von mehreren Containern.<br>Syntax:<br>`docker container pause <CONTAINER> [<CONTAINER> …]`<br>Beispiel:<br>`docker container pause nginx` |
| `docker container restart` | Neustart von einem oder von mehreren gestoppten Containern.<br>Syntax:<br>`docker container restart <CONTAINER> [<CONTAINER> …]`<br>Beispiel:<br>`docker container restart nginx` |
| `docker container unpause` | Beendet die Pause der Prozesse innerhalb von einem oder von mehreren Containern.<br>Syntax:<br>`docker container unpause <CONTAINER> [<CONTAINER> …]`<br>Beispiel:<br>docker container unpause nginx |

| | |
|---|---|
| `docker image ls` | Auflisten aller lokalen Images<br>Syntax:<br>`docker network ls [<OPTIONS>]`<br>Optionen:<br>-f, --format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates. |
| `docker image rm` | Löschen eines oder mehrerer Images aus dem lokalen cash.<br>Syntax:<br>`docker image rm <IMAGE> [<IMAGE> …]`<br>Beispiel:<br>`docker image rm hello-world` |
| `docker kill` | Erzwingt das Beenden von einem oder von mehreren laufenden Containern.<br>Syntax:<br>`docker kill <CONTAINER> [<CON TAINER> …]`<br>Beispiel:<br>`docker kill my_ubuntu` |
| `docker logs` | Abruf der Log Informationen eines Containers.<br>Syntax:<br>`docker logs [<OPTIONS>] <CONTAINER>`<br>Optionen:<br>-f, --follow Erlaubt eine kontinuierliche Ausgabe der Log-Daten.<br>--details Ermöglicht erweiterte Log-Ausgaben<br>--since Zeigt Log-Informationen ab einem bestimmten Zeitpunkt.<br>--until Zeigt Log-Informationen vor einem bestimmten Zeitpunkt.<br>--tail Bestimmt die Anzahl der Zeilen zum Ende der Log-Ausgaben (Standartwert = all)<br><br>Beispiel:<br>`docker logs nginx` |

| | |
|---|---|
| `docker network create` | Erzeugt ein Netzwerk.<br>Syntax:<br>`docker network create [<OPTIONS>] <NAME>`<br>Optionen:<br>-d \| --driver Netzwerktreiber (Standard ist bridge).<br>-ipv6 IPv6 Aktivierung.<br>Beispiel:<br>`docker network create demo_net` |
| `docker network ls` | Auflisten aller Netzwerke<br>Syntax:<br>`docker network ls [<OPTIONS>]`<br>Optionen:<br>-f, --format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates. |
| `docker network connect` | Verbindet einen Container mit einem Netzwerk.<br>Syntax:<br>`docker network connect <NET> <CONTAINER>`<br>Beispiel:<br>`docker network connect demo_net nginx` |
| `docker network disconnect` | Trennt einen Container von einem Netzwerk.<br>Syntax:<br>`docker network disconnect <NET> <CONTAINER>`<br>Beispiel:<br>docker network disconnect demo_net nginx |

| | |
|---|---|
| `docker network inspect` | Zeigt detaillierte Informationen von einem oder mehreren Netzwerken.<br>Syntax:<br>`docker network inspect [OPTIONS] <NET> [<NET> …]`<br>Optionen:<br>-f, --format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates.<br>Beispiel:<br>`docker network inspect demo_net` |
| `docker network rm` | Entfernen eines oder mehrerer Netzwerke.<br>Syntax:<br>`docker network rm <NET> [<NET> …]`<br>Beispiel:<br>`docker network rm demo_net` |
| `docker node inspect` | Zeigt detaillierte Informationen von einem oder mehreren Nodes.<br>Syntax:<br>`docker node inspect [<OPTIONS>] self\|<NODE> [<NODE> …]`<br>Optionen:<br>-f, --format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates.<br>Beispiel:<br>`docker node inspect swarm-manager` |
| `docker node ls` | Auflisten aller Nodes in einem Swarm. |

| | |
|---|---|
| `docker node ps` | Auflisten aller Tasks in einem oder mehreren Nodes. Ohne Angabe eines Nodes wird als Standard der aktuelle Node verwendet.<br>Syntax:<br>`docker node ps [<OPTIONS>] [<NODE> …]`<br>Optionen:<br>--format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates.<br>-f, --filter Erlaubt die Angabe eines Filterkriteriums im Format „key=value“.<br>Beispiel:<br>`docker ps name=nginx worker1` |
| `docker node rm` | Entfernen eines oder mehrerer Nodes von einem Swarm.<br>Syntax:<br>`docker node rm [<OPTIONS>] <NODE> [<NODE> …]`<br>Optionen:<br>-f, --force Erzwingt das Entfernen der Nodes, auch wenn diese aktiv sind.<br>Beispiel:<br>`docker node rm -f swarm-node-02` |
| `docker node promote` | Heraufstufen eines Worker Nodes zum Manager.<br>Syntax:<br>`docker node promote <NODE>`<br>Beispiel:<br>`docker node promote worker1` |
| `docker node demote` | Herabstufen eines Manager Nodes zum Worker.<br>Syntax:<br>`docker node demote <NODE>`<br>Beispiel:<br>`docker node demote worker1` |

| | |
|---|---|
| `docker ps` | Auflisten von Containern.<br>Syntax:<br>`docker ps [<OPTIONS>]`<br>Optionen:<br>-a, --all Anzeige aller Container, auch diejenigen, die nicht aktiv sind.<br>--format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates.<br>-f, --filter Erlaubt die Angabe eines Filterkriteriums im Format „key=value“.<br>Beispiel:<br>`docker ps -a` |
| `docker pull` | Laden eines Images aus der Registry.<br>Syntax:<br>`docker pull <IMAGE> [:<TAG>]`<br>Beispiel:<br>`docker pull hello-world:latest` |
| `docker push` | Hochladen eines Images in die Registry.<br>Syntax:<br>`docker push <IMAGE> [:<TAG>]`<br>Beispiel:<br>`docker push demouser/hello-docker:1.0.1` |
| `docker rm` | Entfernen von einem oder von mehreren Containern.<br>Syntax:<br>docker rm [<OPTIONS>]<CONTAINER> [<CONTAINER> ...]<br>Optionen:<br>-f, --force Erzwingt das Beenden von noch laufenden Containern.<br>Beispiel:<br>`docker rm my_ubuntu` |

| | |
|---|---|
| `docker run` | Starten eines Images als Container.<br>Syntax:<br>`docker run [<OPTIONS>] IMAGE [COMMAND]`<br>Optionen:<br>-d \| --detach Container im Hintergrund ausführen.<br>-i, --interactive Verbindung über STDIO zur Kommunikation geöffnet.<br>-t, --tty Ein pseudo TTY wird instanziiert.<br>-v, --volume Einbinden eines Volumes.<br>--name Zuordnung eines Container-Namens.<br>--network Verbindet den Container mit einem Netzwerk.<br>--rm Container wird nach dem Beenden automatisch entfernt.<br><br>Beispiel:<br>`docker run --name my_ubuntu –it --rm ubuntu`<br>Dieses Beispiel startet einen Container mit dem Namen `my_ubuntu`. Dieser wird aus dem Image `ubuntu:latest` abgeleitet. Der Container wird interaktiv gestartet und verbindet ein Pseudo TTY mit STDIN des Containers. Nach dem beenden des Containers wird dieser gelöscht. |
| `docker search` | Suche nach Images im Docker Hub.<br>Syntax:<br>`docker search <TERM>`<br>Beispiel:<br>`docker search ubuntu` |

| | |
|---|---|
| `docker secret create` | Secret aus einer Datei oder der Standard-Eingabe STDIN erstellen. Wird als letzter Parameter ein Minus ( – ) Zeichen anstelle eines Dateinamens angegeben, so wird STDIN als Eingabemedium verwendet.<br>Syntax:<br>`docker secret create [<OPTIONS>] <SECRET> <FILE>\|-`<br>Optionen:<br>-l \| --label Secret mit Label erstellen.<br>Beispiel:<br>`docker secret create ` `--label rev=202006329 ` `my_secret` |
| `docker secret inspect` | Ausgabe von ausführlichen Informationen über ein oder über mehrere Secrets.<br>Syntax:<br>`docker secret inspect [<OPTIONS>] <SECRET> [<SECRET> ...]`<br>Optionen:<br>-f \| --format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates.<br>--pretty Ausgabe in einem leicht lesbaren Format erzeugen.<br>Beispiel:<br>`docker secret inspect --pretty my_secret` |
| `docker secret ls` | Auflisten aller Secrets in einem Swarm.<br>Syntax:<br>`docker secret ls [<OPTIONS>]`<br>Optionen:<br>-- format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates.<br>Beispiel:<br>`docker secret ls` |

| | |
|---|---|
| `docker secret rm` | Entfernen eines oder mehrerer Secrets.<br>Syntax:<br>`docker secret rm <SECRET> [<SECRET> ...]`<br>Beispiel:<br>`docker secret rm my_secret` |
| `docker service create` | Erzeugen eines neuen Service.<br>Syntax:<br>`docker service create [<OPTIONS>] <IMAGE>`<br>Optionen:<br>--name Zuordnung eines Service-Namens.<br>--network Verbindet den Service mit einem Netzwerk.<br>-t, --tty Ein pseudo TTY wird instanziiert.<br>--mount Einbinden eines Dateisystems.<br>--replicas Anzahl der Tasks.<br>...<br>Beispiel:<br>`docker service create --name ubuntu --replicas=5 ubuntu` |
| `docker service inspect` | Zeigt detaillierte Informationen von einem oder mehreren Services.<br>Syntax:<br>`docker service inspect [<OPTIONS>] <SRVICE> [<SERVICE> ...]`<br>Optionen:<br>-f, --format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates.<br>Beispiel:<br>`docker service inspect ubuntu` |

| | |
|---|---|
| `docker service logs` | Abruf der Log-Informationen eines Service.<br>Syntax:<br>`docker service logs [<OPTIONS>] <SERVICE>`<br>Optionen:<br>-f, --follow Erlaubt eine kontinuierliche Ausgabe der Log-Daten.<br>--details Ermöglicht erweiterte Log-Ausgaben<br>--since Zeigt Log-Informationen ab einem bestimmten Zeitpunkt.<br>--tail Bestimmt die Anzahl der Zeilen zum Ende der Log-Ausgaben (Standartwert = all)<br><br>Beispiel:<br>`docker logs nginx` |
| `docker service ls` | Auflisten aller Services in einem Swarm |
| `docker service rm` | Entfernen eines oder mehrerer Services von einem Swarm.<br>Syntax:<br>`docker service rm <SERVICE> [<SERVICE> …]`<br>Beispiel:<br>`docker service rm ubuntu` |

| | |
|---|---|
| `docker stack deploy` | Erzeugen eines neuen Stacks aus den Informationen einer Konfigurationsdatei im YAML-Format oder Aktualisierung eines existierenden Stacks.<br>Syntax:<br>`docker stack deploy [<OPTIONS>] STACK`<br>Optionen:<br>-c, --compose-file Pfad zur Compose-Datei oder wenn STDIN als Datenquelle genutzt wird.<br>Beispiel:<br>`docker stack deploy --compose-file, docker-compose.yml my_stack` |
| `docker stack ls` | Auflisten aller Stacks in einem Swarm. |
| `docker stack ps` | Auflisten aller Tasks, die im Kontext eines bestimmten Stacks laufen.<br>Syntax:<br>`docker stack ps [<OPTIONS>]`<br>Optionen:<br>--format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates.<br>-f, --filter Erlaubt die Angabe eines Filterkriteriums im Format „key=value“.<br>Beispiel:<br>`docker stack ps -f name=nginx my_stack` |
| `docker stack rm` | Entfernen eines oder mehrerer Stacks.<br>Syntax:<br>`docker stack rm <STACK> [<STACK> ...]`<br>Beispiel:<br>`docker stack rm my_stack` |

| | |
|---|---|
| `docker stack services` | Anzeige der Liste aller Services, die zum Kontext eines bestimmten Stacks gehören.<br>Syntax:<br>`docker stack services [<OPTIONS>] STACK`<br>Optionen:<br>--format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates.<br>-f, --filter Erlaubt die Angabe eines Filterkriteriums im Format „key=value“.<br>Beispiel:<br>`docker stack services my_stack` |
| `docker stop` | Beenden von einem oder von mehreren laufenden Containern.<br>Syntax:<br>`docker stop <CONTAINER> [<CONTAINER> …]`<br>Beispiel:<br>`docker stop my_ubuntu` |
| `docker swarm init` | Initialisieren eines Docker Swarms<br>Syntax:<br>`docker swarm init [<OPTIONS>]`<br>Beispiel:<br>docker swarm init --advertise-addr 192.168.99.121<br>Aktiviert den Swarm-Modus und reagiert auf die angegebene IP-Adresse. |
| `docker swarm join` | Beitritt eines Nodes zu Swarm als Worker oder als Manager<br>Syntax:<br>`docker swarm join [<OPTIONS>] <HOST:PORT>`<br>Beispiel:<br>`docker swarm join --token <token> 192.168.99.121:2377` |

| | |
|---|---|
| `docker swarm join-token` | Ausgabe des Join-Kommandos mit Join Token als Kopiervorlage für den Beitritt eines Nodes zu Swarm als Worker oder als Manager<br>Syntax:<br>`docker swarm join-token worker\|manager`<br>Beispiel:<br>`docker swarm join-token worker` |
| `docker swarm leave` | Als Node eine Swarm verlassen<br>Syntax:<br>`docker swarm leave [--force\|-f]`<br>Optionen:<br>-f, --force Erzwingt das Entfernen der Nodes, Warnungen werden ignoriert.<br>Beispiel:<br>`docker swarm leave --force` |
| `docker volume create` | Erzeugen eines Volumes.<br>Syntax:<br>`docker volume create <VOLUME_NAME>`<br>Beispiel:<br>`docker volume create test_vol` |
| `docker volume inspect` | Zeigt detaillierte Informationen von einem oder mehreren Volumes.<br>Syntax:<br>`docker volume inspect [<OPTIONS>] <VOL> [<VOL> …]`<br>Optionen:<br>-f, --format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates.<br>Beispiel:<br>`docker volume inspect test_vol` |

| | |
|---|---|
| `docker volume ls` | Auflisten aller Volumes.<br>Syntax:<br>`docker volume ls [<OPTIONS>]`<br>Optionen:<br>-f, --format Erlaubt die Formatierung der Ausgabe durch Angabe eines GO Templates. |
| `docker volume rm` | Entfernen eines oder mehrerer Volumes.<br>Syntax:<br>`docker volume rm <VOL> [<VOL> …]`<br>Beispiel:<br>`docker volume rm test_vol` |

Die Tabelle enthält nur eine Auswahl der möglichen CLI-Kommandos. Es werden auch nicht alle Parameter vorgestellt. Sie beschränkt sich auf diejenigen Kommandos und Parameter, welche bei der praktischen Arbeit am häufigsten benutzt werden.

Ausführliche Informationen über alle Docker CLI-Kommandos und deren Parameter mit Beispielen finden Sie auf den offiziellen Internetseiten von Docker:

https://docs.docker.com/engine/reference/commandline/docker/

## 19.11 Format-Angaben für Docker-Kommandos

Viele Docker-Kommandos, wie zum Beispiel das `docker inspect` Kommando, erlauben es, die Ausgabe durch die Angabe des Flags

```
1 --format <FORMAT_STRING> bzw -f <FORMAT_STRING>
```

zu formatieren bzw. zu filtern.

Der Format-String wird dabei im Go-Template-Format angegeben und bezieht sich auf die Datenstruktur, die als Ergebnis eines Kommandos zurückgeliefert wird.

Die Formatangabe wird durch doppelte geschweifte Klammern eingeschlossen.

### 19.11.1 Abfrage der Werte von bestimmten Keys

Das Kommando `docker network inspect none` liefert ohne Parameter –f die folgende, etwas umfangreiche und unübersichtliche Ausgabe:

```
> docker network inspect none
[
    {
        "Name": "none",
        "Id":
"d4c919377363ee4492e84e3f19e1d3d56a5711e7977db6a7c6542c
96fdf49291",
        "Created": "2019-11-11T13:09:39.5236042Z",
        "Scope": "local",
        "Driver": "null",
        "EnableIPv6": false,
        "IPAM”: {
            "Driver”: "default",
            "Options”: null,
            "Config”: []
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

Wenn mich aber nur ein bestimmtes Detail interessiert, dann kann ich die Ausgabe durch das Attribut –f in Verbindung mit einem Format-String filtern.

Beispiele: Im folgenden Beispiel wird mit dem Formatelement `'{{.Name}}'` angegeben, dass aus der obigen Ausgabe nur der Wert des Keys „Name“ herausgefiltert werden soll.

Das Kommando

```
1 docker network inspect -f '{{.Name}}' none
```

liefert als Ergebnis

```
1 none
2
```

Bei verschachtelten Keys werden die Pfadelemente durch Punkt getrennt.

Das Kommando

```
1 docker network inspect -f '{{.IPAM.Driver}}' none
```

liefert damit als Ergebnis

```
1 default
2
```

Durch die Angabe des Schlüsselwortes `json` wird ein Element als JSON-String codiert.

Das Kommando

```
1 docker network inspect -f '{{json .IPAM}}' none
```

liefert hier als Ergebnis

```
1 {"Driver":"default","Options":null,"Config":[]}
```

Zusätzliche Informationen zur Formatierung der Ausgaben von Docker CLI-Kommandos finden Sie auf der folgenden Internetseite von Docker:

https://docs.docker.com/config/formatting/

# Glossar

| Stichwort | Beschreibung |
|---|---|
| Bridge-Netzwerk | Das bridge-Netzwerk ist das Standard-Netzwerk von Docker. Beim Start von Docker wird automatisch das bridge-Netzwerk aktiviert. |
| Build Context | Unter dem Docker „Build Context“ versteht man einen Satz von Dateien, die sich in einem speziellen Verzeichnis befinden oder über eine spezielle URL erreicht werden können. |
| Cache-Speicher | Ein Cache-Speicher ist ein schneller Zwischenspeicher der die Zugriffszeiten auf Daten eines Speichers verkürzen soll. |
| Certificates | Siehe Zertifikate. |
| CLI | Eine Kommandozeilen Schnittstelle (CommandLine Interface) |
| Client | Als Clients bezeichnet man Computersysteme, die Dienste eines Servers Systems nutzen. |
| Cluster | Ein Cluster ist ein Verbund von virtuellen und physikalischen Maschinen zur Steigerung von Rechenleistung und zur Verbesserung der Ausfallsicherheit. |
| Container | Ein Container vereint in sich Software, zusammen mit zugehörigen Bibliotheken, Tools und Konfigurationsdateien. Applikationen laufen so schnell und zuverlässig auf verschiedenen Umgebungen. |
| Container Host | Als Container Host bezeichnet man den Computer der die Container Engine ausführt. |

| | |
|---|---|
| Container Netzwerk | Der Container Host stellt seinen Docker Containern Netzwerke zur Verfügung über die Container miteinander oder mit Client Anwendungen kommunizieren können. |
| Container Registry | Die Container Registry ist eine Serverseitige Anwendung, die es erlaubt, Docker Images zu speichern und bereitzustellen. |
| CURL | Curl steht als Abkürzung für den Namen „Client for URLs“. Es handelt sich hier um ein Kommandozeilen-Tool zur Übertragung von Informationen über eine Internetadresse. |
| Daemon | Ein Daemon ist ein Hintergrundprozess, der anderen Anwendungen, den Clients, verschiedene Dienste zur Verfügung stellt. |
| Deployment | Im Allgemeinen versteht man unter dem Begriff Deployment die automatisierte Verteilung, Installation, Konfiguration und Wartung von Software auf mehreren Computersystemen. |
| Detached Mode | Werden Docker Container Im Detached Mode ausgeführt, so laufen diese im Hintergrund. |
| Digest | Unter Docker stellt ein Digest eine Folge von hexadezimalen Zeichen dar, die ein Docker Objekt eindeutig identifiziert. |
| DNS-Server | DNS-Server haben die Aufgabe einer URL die richtige IP zuzuweisen oder einer IP die richtige URL. DNS steht dabei für “Domain Name System“. |
| Docker Compose | Docker Compose ist ein Tool, das zum Definieren und Freigeben von Multicontaineranwendungen entwickelt wurde. Mit Compose können Sie eine YAML-Datei erstellen, um die Dienste zu definieren, die Sie mit einem einzigen Befehl starten bzw. beenden können. |

| | |
|---|---|
| Docker Configs | Docker Configs bieten die Möglichkeit, unkritische Informationen, wie zum Beispiel HTML-Seiten, zu speichern, ohne dass Konfigurationsdateien in den Images für Container eingebunden werden müssen oder Informationen über Umgebungsvariablen zur Verfügung gestellt werden. |
| Docker Desktop | Bei ‚Docker Desktop' handelt es sich um Applikationen für Windows und MacOS, mit deren Hilfe recht einfach und komfortabel fertige Container-Anwendungen erstellt werden können. Dabei können beliebige Frameworks, Programmiersprachen und Zielplattformen zum Einsatz kommen. |
| Docker Engine | Die Docker Engine stellt die Laufzeit Umgebung für Container zur Verfügung und läuft auf Linux, macOS und Windows Server Betriebssystemen. |
| Docker File | Das Dockerfile ist eine Textdatei welche Linux Kommandos enthält die ein Anwender auch auf der Linux Kommandozeile eingeben könnte. Im Dockerfile erledigen diese Kommandos alle Aufgaben die nötig sind um ein Docker Image zusammenzustellen. |
| Docker Hub | Der Docker Hub ist ein auf Cloud-Technologie basierter Repository Service, den Docker-Anwender und -Partner nutzen können, um Container Images abzulegen und zu verwalten. |
| Docker Service | Die Kombination der Tasks, welche auf einem Manager oder Worker Node laufen, nennt man bei Docker Swarm Service. Dabei wird spezifiziert, aus welchem Container Image die Tasks aufgebaut werden und welche Kommandos zur Laufzeit innerhalb eines Containers ausgeführt werden. |

| | |
|---|---|
| Docker Stack | Mit Docker Stack ist es möglich, mehrere Docker Services zu Multi-Container-Applikationen zu verknüpfen. |
| Docker Swarm | Bei Docker Swarm handelt es sich um ein Orchestrierungs-Tool. Diese Art von Tools dienen der unterstützen bei der Verwaltung von verteilten Systemen die aus zahlreichen Containern bestehen, welche wiederum über viele Host-Rechner verteilt sein können. |
| Docker Volume | Docker Volumes sind Dateisysteme die in Docker Container montiert werden um dort Daten dauerhaft, also nicht flüchtig, zu speichern |
| Gateway | Als Gateway bezeichnet man eine Hard- oder auch Softwarekomponente, welche die Netzwerkverbindung zu anderen Systemen außerhalb herstellt. |
| Host-Netzwerk | Das host-Netzwerk verbindet einen Container mit dem hosteigenen Netzwerk. Dadurch wird ein Container direkt an die IP-Adresse des Container Hosts angebunden. |
| Hyper-V | Hyper-V ist eine Virtualisierungssoftware von Microsoft. Hiermit können verschiedene Betriebssysteme auf Ihrem PC installiert und gleichzeitig genutzt werden. |
| Image | Container Images werden genutzt um zur Laufzeit Container Instanzen zu erzeugen. Bei Docker werden Docker Images zu Docker Containern, wenn sie auf einer Docker Engine als Prozess ausgeführt werden. |
| IP-Adresse | Deine IP-Adresse ist eine Zahlenkombination über die jedes System in einem Netzwerk eindeutig identifiziert werden kann. |
| JSON | JSON (JavaScript Object Notation) ist ein textbasiertes Datenformat und wird zum Austausch von Informationen zwischen Anwendungen eingesetzt. |

| | |
|---|---|
| Kubernetes | Kubernetes ist, so wie Docker Swarm auch, ein Tool zur Orchestrierung von containerbasierten Anwendungen. Es handelt sich hier um ein System zur Automatisierung, Bereitstellung, Skalierung und Verwaltung von großen Container-Applikationen. |
| Kubernetes Master | Ein Kubernetes Master bildet die Kontrollebene eines Kubernetes Clusters und ist für dessen Organisation und Koordination<br><br>verantwortlich. |
| Kubernetes Node | Kubernetes Nodes sind in einem Cluster für die eigentliche Funktion einer Applikation verantwortlich. |
| Kubernetes Services | Kubernetes Services sind Instanzen die den Client-Applikationen eine einheitliche Schnittstelle zu Kubernetes Pods innerhalb eines Deployment bieten. |
| Load Balancing | Beim Load Balancing geht es um die gleichmäßige Verteilung der Gesamtlast auf parallel arbeitende Tasks. |
| MAC-Adresse | Die MAC-Adresse (Media-Access-Control-Adresse oder Physikalische Adresse) ist die Hardware-Adresse jedes einzelnen Netzwerkgerätes. Sie dient als weltweit eindeutiger Identifikator von Geräten in Rechnernetzen. |
| Manager Nodes | Manager Nodes sind dafür verantwortlich, dass die Aufgaben, welche als Tasks bezeichnet werden, den Worker Nodes zugeteilt werden. Manager Nodes sind darüber hinaus für die Orchestrierung und das Management eines Clusters verantwortlich. Sie müssen in diesem Zusammenhang sicherstellen, dass der festgelegte Status eines Swarm immer aufrechterhalten wird. |

| | |
|---|---|
| Microservice | Microservices sind ein architekturbezogener und organisatorischer Ansatz in der Softwareentwicklung, bei dem Software aus kleinen unabhängigen Services besteht, die über sorgfältig definierte APIs kommunizieren. |
| Node | Ein Node (oder auch Knoten) ist eine Instanz einer Docker Engine, auf welcher der Swarm Mode aktiviert ist und die zu einem Docker Swarm gehört. |
| Orchestrierung | Als Orchestrierung bezeichnet man die flexible und automatisierte Kombination, Konfiguration und Koordination verschiedener Computer und deren Dienste zu einem Gesamtsystem. |
| Overlay-Netzwerk | Overlay- Netzwerke können Docker Container verbinden, welche auf mehreren Host-Rechnern verteilt sind. |
| PHP | PHP steht für PHP-Hypertext Preprocessor. Es ist eine Scriptsprache, die vor allen zum Erstellen von interaktiven und dynamischen Web-Applikationen eingesetzt wird |
| Port | Ein Port ist Bestandteil einer Netzwerkadresse und dienen dazu verschiedene Verbindungen zwischen einem Paar von Endpunkten zu identifizieren. |
| Pods | Pods bilden die kleinste Einheit von Kubernetes. Docker Container, wie auch Container aus anderen Systemen, können nicht direkt in einem Kubernetes Node ausgeführt werden. Dafür ist so etwas wie ein Adapter nötig. Diese Adapterfunktion erfüllen Pods. |
| Prompt | Ein Prompt ist ein Zeichen in einer Kommando Shell das dem Benutzer anzeigen soll, dass dort ein Befehl eigegeben werden kann. |

| | |
|---|---|
| Proxy | Ein Proxy ist ein System in einem Netzwerk aus Computern das als Stellvertreter bzw. als Vermittler arbeitet. Es nimmt Anfragen entgegen und leitet diese an andere Systeme weiter. |
| Raft Datenbank | Im Swarm-Modus wird von allen Manager Nodes eine eigene Instanz einer Datenbank integriert, durch welche der globale Status aller Cluster verwaltet wird. Dieses Datenbank-System arbeitet mit dem sogenannten „Raft Distributed Consensus Algorithmus". Dieser Algorithmus stellt sicher, dass der Inhalt aller Datenbankinstanzen, auf die in den Manager Nodes zugegriffen wird, konsistent ist. |
| Registry | Eine Docker Registry speichert und verwaltet Docker Images. |
| Repository | Ein Repository ist ein verwaltetes Archiv zur Speicherung und Beschreibung digitaler Objekte. |
| REST | Die Abkürzung REST steht für Representational State Transfer. Ein REST API definiert eine Programmierschnittstelle, die beschreibt, wie verteilte Systeme miteinander kommunizieren können. |
| Scheduler | Ein Scheduler regelt die Organisation und die zeitliche Ausführung mehrerer Aufgaben (Tasks) in einem System. |
| Secrets | In der Docker-Welt spricht man von „Secrets" (Geheimnissen), wenn es sich um schützenswerte oder sicherheitskritische Daten handelt, wie zum Beispiel Passwörter, Schlüssel, Zertifikate oder Ähnliches. |
| Server | Ein Server ist ein Computersystem, welches Funktionen, Dienste, Daten und Ressourcen für andere Systeme, sogenannte Clients, bereitstellt. |

| | |
|---|---|
| Shell | Bei einer Shell handelt es sich um ein Programm, mit dessen Hilfe ein Anwender mit einem Computersystem interagieren kann. In einer Kommando Shell kann man zum Beispiel Systemkommandos in Textform eingeben und ausführen lassen. |
| Skalieren | Unter dem Begriff Skalieren versteht man bei Systemen, die Aktionen die nötig sind um deren Größe bzw. Ressourcen an die aktuellen Anforderungen anzupassen. |
| SQL | SQL (Structured Query Language) ist eine Standardsprache zur Bearbeitung von strukturierten Datenbanken. |
| Syntax | Eine Syntax ist ein Regelwerk das bestimmt, wie verschiedene Zeichen in einer Programmiersprache zusammengesetzt werden müssen. Es handelt sich sozusagen um die Grammatik von Programmiersprachen |
| Tag | Der Begriff „Tag“ kommt aus dem Englischen und bedeutet Etikett oder Schlagwort. In der Informatik werden mit Tags Datenelemente klassifiziert und strukturiert. |
| Task | Ein Task ist die kleinste ausführbare Einheit eines Docker Swarm. Ein Task beinhaltet die Instanz eines Docker Containers mit den Kommandos, welche die servicespezifischen Aufgaben erledigen. Manager Nodes weisen Tasks den Worker Nodes in einer festgelegten Anzahl von Instanzen zu. |
| URI / URL | Mit dem (Uniform Resource Identifier) lassen sich unterschiedliche Ressourcen über das Internet ansprechen. Die häufigste Form des URI ist der Uniform Resource Locator (URL), also die Internetadresse eines Web Auftritts. |

| | |
|---|---|
| Virtual Switch | Ein virtueller Switch (Virtual Switch) ist eine Software, über die virtuelle Maschinen (VMs) mit anderen kommunizieren können. Wie ein pyhsikalischer Ethernet-Switch, leitet ein virtueller Switch Datenpakete weiter. |
| Worker Nodes | Worker Nodes führen die Tasks aus, welche ihnen von den Manager Nodes zugeteilt worden sind. Standardmäßig können auch Manager Nodes zusätzlich die Funktionalität von Worker Nodes übernehmen. |
| YAML | YAML ist eine von XML abgeleitete Auszeichnungssprache die hautsächlich dazu dient Datenstrukturen zu definieren. |
| Zertifikate | Ein Zertifikat ist ein digitaler Datensatz, der bestimmte Eigenschaften von Personen oder Objekten bestätigt und dessen Authentizität und Integrität durch kryptografische Verfahren geprüft werden kann.<br>Bei zertifikatbasierter Authentifizierung wird ein digitales Zertifikat (Certificate) verwendet, um eine Entität (einen Benutzer, ein Gerät oder ein System) zu identifizieren, bevor der Zugriff auf eine Ressource, ein Netzwerk oder eine Anwendung gewährt wird.<br>Zertifikate werden durch eine offizielle Zertifizierungsstelle, die Certification Authority (CA), erstellt. |

# Index

## *U*



## *V*



## *W*



## *Y*