c't KUBERNETES

Container orchestrieren in der Praxis

Docker und Podman im DevOps-Alltag

Weg von Docker – hin zu Podman
Security: Container-Images scannen

Kubernetes-Praxis für Container-Profis

Schlanke Cluster On-Premises und in der Cloud

Strategien für Storage, Netzwerk und Security

Erprobte Konzepte statt Anfängerfehler

GitOps: Automatische Clusterverwaltung mit Helm und Argo CD

Alles redundant: Storage mit Longhorn konfigurieren

€ 22,50
CH CHF 33.90
AT € 24,50
LUX € 25,90

4 197665 622503 02

Wir suchen

Senior Open Source Admins/Consultants

(w|m|d)

Das sind Deine Stärken:

- eine schnelle Auffassungsgabe
- analytisches Denken
- Leidenschaft für Linux/Open Source
- Selbstorganisation & Kommunikation

Das erwartet Dich bei uns:

- 4-Tage-Woche & überwiegend Homeoffice
- vielfältige & abwechslungsreiche Einsätze
- familiäres Klima & flache Hierarchien
- Zusatzversicherungen & Jobrad

Das bringst Du mit:

- Erfahrungen mit Linux
- Kenntnisse in einem oder mehreren der folgenden Bereiche wünschenswert:
  - Cloud
  - Container
  - System- und Konfigurationsmanagment
  - High Availability
  - Monitoring
  - Continuous Integration/Delivery

Mehr erfahren & bewerben:
jobs@b1-systems.de
Formlose Bewerbung genügt

B1 SYSTEMS

B1 Systems GmbH - Ihr Linux-Partner

Linux/Open Source Consulting, Training, Managed Service & Support

ROCKOLDING · KÖLN · BERLIN · DRESDEN · JENA

www.b1-systems.de · info@b1-systems.de

Editorial

Kubernetes: Der Heilsbringer für Containerprobleme?

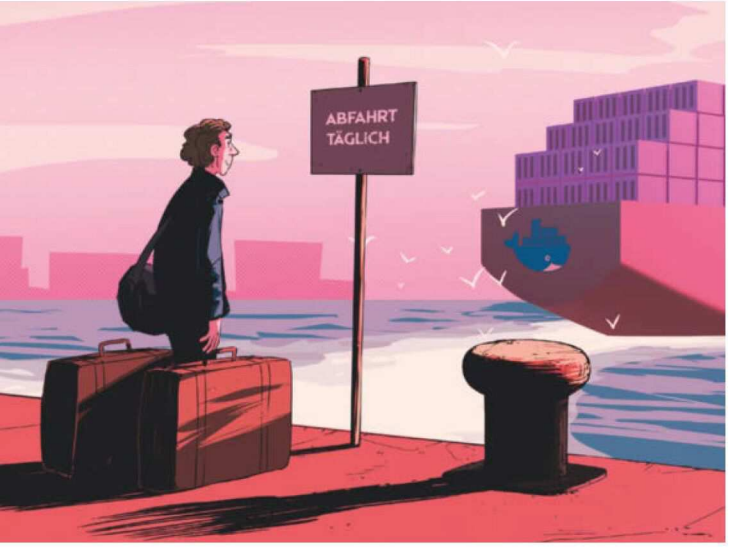
Es ist ein Running Gag in Teilen der IT geworden: Virtualisierung löst unsere Probleme mit Computern. Container lösen unsere Probleme mit Virtualisierung. Und wer löst unsere Probleme mit Containern? Kubernetes vielleicht? Genau mit diesem Ziel ist der Containerorchestrator im Jahr 2015 angetreten, um Admins beim Steuern riesiger Containerumgebungen in Cloudrechenzentren etwa bei Google, AWS & Co. zu unterstützen. Doch diese Lage hat sich gewandelt, mittlerweile hilft Kubernetes auch Mittelständlern.

Wenn Sie vor der Herausforderung stehen, die Containeraufgaben Ihrer Organisation mit Docker, Podman oder direkt mit Kubernetes zu lösen, haben Sie das richtige Heft in der Hand. Es richtet sich vornehmlich an alle, die schon mit Containern arbeiten und Erfahrungen gesammelt haben, an Admins wie Entwickler gleichermaßen. Sie erfahren Schritt für Schritt, wie Sie Ihren ersten Kubernetes-Cluster einrichten und mit YAML-Dokumenten konfigurieren, was es mit Pods, Services, IngressRoutes und PersistentVolumeClaims auf sich hat. Wir reichen Ihnen das komplette Handwerkszeug, um eine containerisierte Umgebung, die bisher in Docker oder Podman lief, in Kubernetes zu überführen. Weiter geht es mit erprobten Strategien aus der Praxis für Storage und einen automatisierten Containerbetrieb auf Basis eines Git-Repos.

All Ihre Probleme mit Computern im Allgemeinen und Containern im Speziellen werden Sie damit nicht lösen, aber mit Kubernetes haben Sie Zugriff auf ein mächtiges Werkzeug. Das Wissen dazu ist die Eintrittskarte in ein überwältigendes Open-Source-Ökosystem etwa für Monitoring über redundante Datenbanken und automatische Skalierung bei Lastspitzen. Fast jede Hürde, auf die Sie in Ihrer Kubernetes-Karriere stoßen, hat schon jemand anderes vor Ihnen genommen. Und in den meisten Fällen gibt es ein Open-Source-Projekt, das Ihnen herüberhilft.

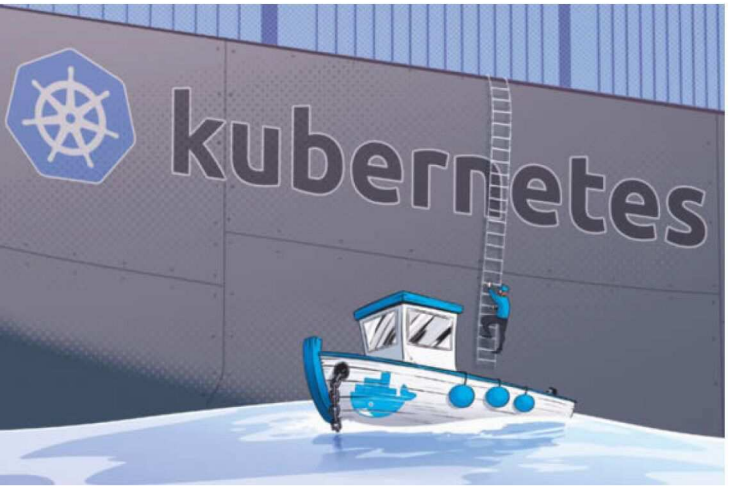
Jan Mahn

Inhalt



CONTAINER MIT DOCKER UND PODMAN

Docker und Podman sind auf Entwicklerrechnern und kleinen Servern zu Hause und die Grundlage für jedes Container-Projekt. Der Ein- und Umstieg ist einfach; bei Security und Netzwerkverbindungen gilt es, auch unbekanntere Funktionen zu entdecken.



DER KUBERNETES-LERNPFAD

Sobald die Anforderungen steigen, reichen Docker und Podman nicht aus – das Wissen können Admins und Entwickler in die Kubernetes-Welt mitnehmen und steigen somit Stück für Stück in die Technik ein, die auch Dienste von Weltkonzernen betreibt.

KONZEPTE FÜR FORTGESCHRITTENE

Der erste Cluster läuft, die ersten Container sind umgezogen – Zeit für einen Blick auf erprobte Konzepte, die den Admin-Alltag leichter machen. Und auf Open-Source-Software, die alltägliche Probleme aus der Welt schafft.

ZUM HEFT

heise Academy-Aktion:

Schneller Einstieg in Podman

Das Open-Source-Tool Podman verspricht hohe Sicherheit und mehr Zugriffsmöglichkeiten für das moderne Container-Management. Im Videokurs der heise Academy unternehmen Sie eine spannende Tour durch die Grundlagen dieser vollwertigen Engine – mit der Expertise der Entwickler.

Von **Markus Richter**

Sie beschleunigen die Entwicklung von Anwendungen und vereinfachen sie zugleich: Container sind zu Recht einer der großen Trends der professionellen IT. Engineer Teams von Red Hat haben dafür zusammen mit der Open Source Community den POD-Manager Podman auf der Basis von Docker entwickelt und eine Lösung geschaffen, mit der sich das gesamte Container-IT-Ökosystem verwalten lässt. Im Unterschied zu anderen Tools wird Podman ohne Daemons ausgeführt, die eine Sicherheitsschwachstelle darstellen können. Auch auf Root-Rechte verzichtet man bei Podman.

Die heise Academy hat für ihren Einführungsvideokurs zu Podman Valentin Rothberg gewinnen können, der als Software Engineer bei Red Hat unmittelbar mit der Materie vertraut ist. Mit dem Kauf dieses c't-Sonderheftes können Sie gegen eine Schutzgebühr von 4,95 Euro statt regulär 49 Euro an dem Videokurs teilnehmen. Der Trainer stellt darin die wichtigsten Basics von Linux-Containern und Container-Images vor. Wie sehen Container-Prozesse auf dem Linux-System aus? Woraus besteht ein Container-Image? Für die Arbeit mit lokalen Daten und Verzeichnissen lernen Sie, wie Sie mit Volumes und Mounts umgehen, insbesondere als Rootless-Nutzer. Mit Podman können Sie bequem auch auf dem Mac- oder Windows-Rechner mit Containern arbeiten.

Academy-Trainer Valentin Rothberg ist in Red Hat's „Container Runtimes Team" beschäftigt und kennt sich umfassend mit Container-Werkzeugen

Über heise.de/s/VxVd erhalten Sie diesen Videokurs mit dem Rabattcode KUBERNETES2023 einmalig für nur 4,95 Euro, statt für 49 Euro*.

*Preis- und andere Irrtümer vorbehalten. Das Angebot ist gültig bis zum 31.12.2023 (Stand: Juni 2023).

Das lernen Sie im Videokurs

- Einen Linux-Container verwenden
- Die Engine Podman verstehen
- Container mit Podman ausführen und verwalten
- Arbeiten mit Volumes und Mounts
- Rootless-Container nutzen

und ihren Technologien aus. Er hat zu vielen anderen Projekten in der Container-Landschaft beigetragen, darunter Kubernetes, Linux-Kernel, Moby, Google Cloud und container-diff. Vor seiner Tätigkeit in der Industrie war Valentin Rothberg in der Forschung und akademischen Lehre von Betriebssystemen tätig.

Die heise Academy – IT-Weiterbildung neu gedacht

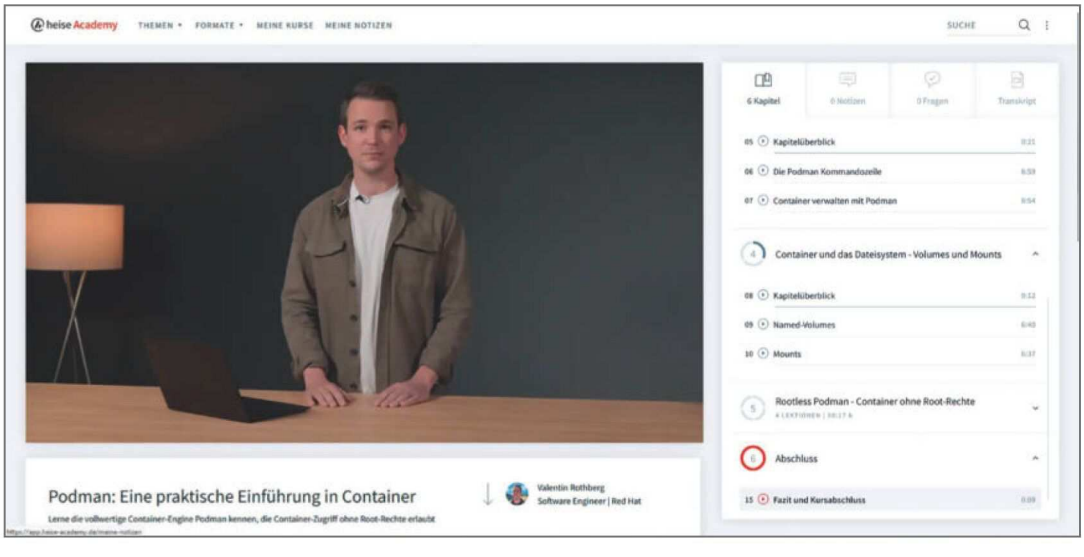
Dieser Videokurs gehört zum Angebot der heise Academy, die sämtliche Weiterbildungen der heise-Verlagsgruppe unter einem Dach bündelt. Interessierte IT-Professionals und Unternehmen finden auf www.heise-academy.de zeitgemäße und maßgeschneiderte Wissensangebote. Damit können Sie Ihre Skills vertiefen, neue Schwerpunkte in Ihrer Arbeit setzen, Ihre Karriere voranbringen und mit Spaß lernen.

Mit dem Launch der neuen Seite sind ab sofort alle Weiterbildungsangebote übersichtlich und strukturiert auf einer Plattform vereint. Finden Sie Ihren passenden On-Demand-Kurs für das selbstbestimmte Lernen, oder besuchen Sie von führenden IT-Experten durchgeführte Live-Events wie Webinare, Konferenzen, Schulungen und Workshops.

Von Netzwerken und Systemen über IT-Projektmanagement, Softwareentwicklung, Data Science und IT-Security bis hin zu Web- und Cloud-Technologien bietet die heise Academy jede Menge geballtes Fachwissen für die professionelle Anwendung. Dabei steht die Wissensvermittlung durch ausgewählte Experten im Mittelpunkt. Mehr als 100 erfahrene Trainer kommen aus dem gesamten deutschsprachigen Raum und bedienen unterschiedliche Schwerpunkte mit starkem Praxisbezug.

Buchen Sie einzelne Events oder Kurse oder treten Sie dem Campus mit dem Academy Pass bei. Diese Lern-Flatrate beinhaltet mehr als 100 Videokurse – darunter auch weitere zum Thema Container, Kubernetes und Podman – sowie eine Vielzahl an Webinaren. Die Videokurse schauen Sie in einem eigens entwickelten Player, der eine komfortable Benutzeroberfläche bietet. Durchsuchen Sie den Kurs mithilfe einer Volltextsuche nach Stichworten, hinterlegen Sie persönliche Notizen und stellen Sie Fragen an die Trainer. So ist neben den Vorteilen der On-Demand-Nutzung die Möglichkeit zur Interaktion mit den Experten gegeben. Der Campus bietet zudem mehr als 100 Live-Webinare jährlich, die immer am Puls der Zeit sind. Hier wird über Trendthemen diskutiert, an praktischen Fallbeispielen geübt und eine Lösung für jedes IT-Problem angeboten.

Alle Infos zu den Angeboten finden Sie unter www.heise-academy.de. (anm) c't



Red-Hat-Entwickler Valentin Rothberg zeigt im Videokurs die Grundlagen von Podman.

Container verstehen und loslegen

Ab 2013 hat Docker die Containertechnik salonfähig gemacht. Von Kritikern erst als kurzfristiger Hype verschrien, sind Container mit Docker, Podman und Kubernetes wenig später zur etablierten Technik geworden. Für den Einstieg ist es lange noch nicht zu spät: wie Sie von Containern profitieren und Software mit und ohne Docker betreiben.

Von **Jan Mahn**

Bild: Albert Hulm

Technische Neuerungen spalten oft die Gemüter: Auf der einen Seite gibt es die Early Adopter, die alles Neue sofort anfassen und ausprobieren müssen und mit dem Risiko leben, wenig später ein totes Pferd reiten zu müssen. Auf der anderen Seite stehen die Skeptiker, die sich das bunte Treiben lieber von außen anschauen und darauf warten, dass sich eine gewisse Stabilität einstellt – oft ist das eine gute Strategie, weil viele Hype-Themen nach wenigen Jahren still und leise in der Versenkung verschwinden. Auch bei Docker, eine Software, die 2013 erstmals erschien, war Vorsicht durchaus angebracht. Dennoch handelten viele die Technik eines kleinen Open-Source-Start-ups schnell als Quasi-Industriestandard.

Viele Baustellen im Docker-Unterbau haben sich rasanter verändert, als die Entwickler ihre Dokus und die Vertriebler ihre Prospekte anpassen konnten. Nicht nur die Open-Source-Software Docker selbst, auch das Geschäftsmodell der Docker Inc. hat einige Kurswechsel hinter sich.

Wer bisher abgewartet hat, konnte sich einige Sackgassen und Irrwege ersparen. Nach fast 10 Jahren auf dem Markt zeichnet sich aber ab: Container bleiben uns erhalten, ob mit oder ohne Docker. Und die Zeit der großen Umbrüche ist vorbei. Auf den Maschinen von Entwicklern laufen Docker oder Podman, um Images zu erproben, zu entwickeln. Auch in kleinen Produktivumgebungen sind Docker und Podman zu Hause.

Sobald die Anforderungen größer werden, kommen die Container in einen Kubernetes-Cluster. Der Umstieg ist vergleichsweise einfach, weil dieselben Images zum Einsatz kommen.

Seit etwa 2016 berichten wir in c't regelmäßig über Container-Technik im Allgemeinen und Docker im Speziellen, veröffentlichen Artikel mit Grundlagen und stellen viele Projekte auf Container-Basis vor, die Docker-Grundwissen voraussetzen. Häufig erreicht uns daher die Frage, welche schon veröffentlichten Artikel man lesen muss, um schnell ins Thema einzusteigen. Das ist nicht ganz so einfach: Weil sich vieles in der Containerwelt innerhalb kürzester Zeit weiterentwickelt hat, können wir oft schon zwei Jahre alte Artikel nicht mehr guten Gewissens empfehlen. Zeit, den Stand des Container-Universums nachzuzeichnen. Für alle, die sich Containern bisher verweigert haben, die eine gewisse Reife der Software abwarten wollten, oder die jetzt das Gefühl beschleicht, trotz Docker-Erfahrung in den letzten Jahren wichtige Neuerungen verpasst zu haben. Der Artikel beschränkt sich auf Container mit Linux-Unterbau – Windows-Container auf Windows-Servern sind eine eigene Baustelle. Wenn Sie Docker den Rücken kehren wollen und stattdessen einen Blick auf den Open-Source-Marktmitbewerber Podman werfen wollen, werden Sie im Artikel „Von Docker Desktop auf Podman wechseln“ auf Seite 28 fündig. Fast alles, was unter Docker funktioniert, klappt auch mit Podman.

Was habe ich davon?

Als reiner Desktop-PC-Anwender (sei es unter Linux, Windows oder macOS) haben Sie nichts verpasst, wenn die Dockerei bisher an Ihnen vorbeiging. Docker ist eine Software, die für Admins und Entwickler gemacht wurde. Sinnvolles Einsatzgebiet sind Serverdienste, aber auch für Kommandozeilenwerkzeuge können Container durchaus nützlich sein. Mit etwas Bastelei kann man theoretisch auch grafische Anwendungen im Container betreiben, erste Wahl für Desktop-Software sind die Container aber nicht.

Um den Nutzen von Containern und die Funktionsweise zu verstehen, ohnt ein Blick auf die Arbeitsschritte, die ohne nötig sind, bis eine Serversoftware funktioniert. Anschauliches Beispiel ist die populäre Bloganwendung WordPress auf einem Linux-Server. Damit WordPress läuft, braucht man einen Ordner mit den WordPress-Dateien selbst, eine PHP-Engine, einen Webserver (Apache, Nginx …) und eine SQL-Datenbank (MariaDB oder MySQL).

Auf einem nackten Linux-Server würde man zur Installation damit beginnen, die Komponenten über den Paketmanager herunterzuladen, etwa per `apt install` in Debian, Mint oder Ubuntu. Anschließend konfiguriert man alle Bausteine einzeln in ihren Konfigurationsdateien (die unter Linux meist im Ordner /etc. liegen) und verdrahtet dann die Komponenten miteinander: Der Webserver muss mit der PHP-Engine sprechen und ihm Dateien mit der Endung .php zum Parsen vorwerfen, der PHP-Code muss die Datenbank erreichen und wissen, wo er hochgeladene Bilder im Dateisystem ablegen soll. Außerdem braucht der Webserver noch Regeln, damit hübsche URLs richtig übersetzt werden und WordPress die richtige Seite anzeigt. Bis man eine gute Schritt-für-Schritt-Anleitung zum Installieren von WordPress per Hand durchgespielt hat, vergehen etwa 30 Minuten.

Das große Problem mit solchen Installationen bemerkt man spätestens, wenn man auf die Idee kommt, auf dem Server mit der öffentlich zugänglichen Website noch eine Testumgebung einzurich-

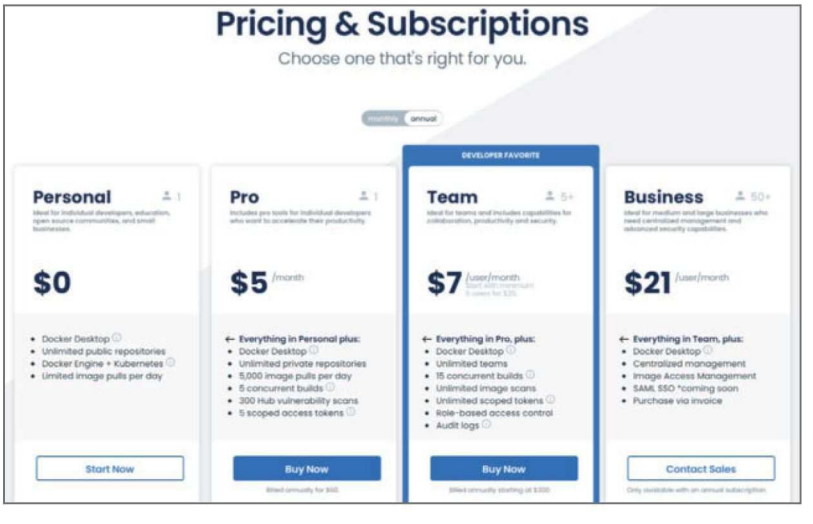
ten. In einer solchen möchte man typischerweise am Layout arbeiten, oder neue Versionen von WordPress, PHP oder der Datenbank ausprobieren. Auf derselben Maschine kann man solche Experimente meist vergessen. Schuld daran sind Abhängigkeiten und hartkodierte Pfade zu Konfigurationsdateien. Viele Anwendungen sind schlicht nicht darauf ausgelegt, dass in einem Betriebssystem mehrere Versionen parallel installiert sind. Schon zwei MariaDB-Instanzen verschiedener Versionen auf einer Maschine sind mit erheblichem Einrichtungsaufwand verbunden.

In ganz ferner Vergangenheit blieb einem nichts anderes übrig, als Hardware für einen zweiten Server anzuschaffen. Besser wurde das erst mit virtuellen Maschinen (VMs) – ein großer Fortschritt, weil man Hardware endlich ausreizen und mehrere Betriebssysteme parallel auf einer Maschine booten konnte. Hat man sich für das WordPress-Beispiel ein Installationsskript gebaut, das alle Einrichtungsschritte durchspielt, könnte man vergleichsweise zügig eine Testumgebung in einer virtuellen Maschine hochfahren und die virtuelle Festplatte auch mit Admin-Kollegen und Entwicklern teilen. Virtualisieren hat aber einen entscheidenden Nachteil: Jede VM bootet einen Kernel des eingesetzten Betriebssystems. Das blockiert, ohne dass eine Anwendung liefe, schon mal etwa ein Gigabyte Arbeitsspeicher (unter Windows Server etwas mehr). Außerdem blockiert eine VM recht viel Festplattenspeicher, weil auf der virtuellen Festplatte ein komplettes Dateisystem mit dem Betriebssystem liegt.

Containertechnik wird immer wieder mit Virtualisierung in einen Topf geworfen, unterscheidet sich aber grundsätzlich von ihr: Ein Container ist keine virtuelle Maschine, sondern ein gewöhnlicher Prozess, dem eine für ihn optimale Scheinwelt vorgegaukelt wird. Für einen Container wird kein Kernel gebootet, der Prozess im Container läuft mit dem Kernel des gastgebenden Systems, beansprucht daher nur so viel RAM wie der Prozess auch ohne Container bräuchte. Die Container-Runtime (eine solche steckt in Docker) ist dafür verantwortlich, die Scheinwelt aufrechtzuerhalten: Der Prozess im Container bekommt ein virtuelles Dateisystem vorgesetzt, in dem nur das existiert, was er braucht. Also die Anwendung selbst und alle Abhängigkeiten – in genau der richtigen Version.

Vom Rest des Betriebssystems sieht der Prozess nichts, keine anderen Prozesse und auch keine anderen Dateien. Unter Linux arbeitet Docker dabei mit einer Kernel-Funktion names Cgroups, die schon lange vor Docker erfunden wurde. Ganz von der Außenwelt abgeschnitten ist der Prozess im Container aber nicht, dann wäre er ja für nichts zu gebrauchen. Die Container-Runtime kann ihm zum Beispiel eine virtuelle Netzwerkkarte vorsetzen, über die er mit der Außenwelt sprechen kann.

Durch die Kapselung der Prozesse in ihrer maßgeschneiderten Hülle kann man problemlos alles Mögliche parallel auf einer einzigen Maschine mit installiertem Docker Daemon (einen solchen Server nennt man auch Docker-Host) betreiben: eine WordPress-Instanz mit PHP 7, eine weitere mit PHP 8, eine



Seit der letzten Änderung des Geschäftsmodells müssen Nutzer von Docker Desktop zahlen, wenn sie in Unternehmen mit mehr als 250 Mitarbeitern arbeiten und dort Docker einsetzen. Der Docker Daemon für Server bleibt aber Open Source und kostenlos.

Datenbank auf MariaDB- und eine auf MySQL-Basis. Weil kein RAM für Virtualisierung verschwendet wird, laufen all diese Container auch problemlos nebeneinander auf einem Entwickler-Notebook mit überschaubarer Ausstattung. Das ist ein durchaus gängiges Szenario: Auf der lokalen Maschine probiert man eine Zusammenstellung von Containern aus und startet sie dann auf dem Server. Von wenigen Ausnahmen abgesehen, verhält sich ein und derselbe Container überall gleich.

Durch die Kapselung verlieren auch Anwendungen ihren Schrecken, die in fremden Programmiersprachen geschrieben sind. Wer etwa um die Java-Welt bisher einen Bogen gemacht hat, kann Java-Anwendungen problemlos im Container ausführen, ohne sich damit befassen zu müssen, wie die Java-Runtime auf die Maschine kommt und was eine Java-Runtime überhaupt ist. Man braucht lediglich ein Container-Abbild, in dem Java und die Anwendung stecken. Auch das Aufräumen ist leicht: Stoppt und löscht man einen solchen Container, bleibt auf dem Docker-Host nichts zurück, weil Docker das für den Container erschaffene Dateisystem komplett entsorgt.

Abbilder erzeugen

Erfolgreich wurde Docker, weil es nicht nur die gesamte Arbeit mit Low-Level-Techniken rund um Cgroups & Co. vom Nutzer fernhält, sondern auch den Umgang mit Container-Abbildern sehr einfach gemacht hat. Solche Abbilder (Images) enthalten das Dateisystem für den Container, verpackt in Tar-Archiven. Beim Start packt der Docker Daemon sie aus und setzt sie dem Container vor. Ein WordPress-Container mit Apache könnte zum Beispiel die Ordnerstruktur mit den WordPress-Dateien enthalten, außerdem die Binärdatei von Apache, Konfigurationsdateien und PHP. Was es nicht enthält, ist die SQL-Datenbank, auch wenn man diese für den Betrieb von WordPress braucht. Doch mehrere Prozesse in einem Container widersprechen der Docker-Philosophie: Jeder Container führt genau einen Prozess aus, in diesem Fall ist das der Webserver Apache. Die Datenbank gehört also unbedingt in einen weiteren Container.

Container-Abbilder knüpft man nicht per Hand zusammen, indem man alle Abhängigkeiten selbst in eine Tar-Datei verpackt. Stattdessen schreibt man eine Rezeptdatei, das Dockerfile. Dieses Rezept übergibt man dem Befehl `docker build`. Das Dockerfile ist eine gleichnamige Textdatei (ohne Dateiendung und mit einem Großbuchstaben am Anfang). Jede Zeile des Dockerfile beginnt mit einer Anweisung in Großbuchstaben, das Rezept für einen kleinen Container, der eine Website mit Nginx ausliefert, sieht zum Beispiel folgendermaßen aus:

```
FROM nginx:alpine
COPY index.htm /var/www/html/
```

Ein Dockerfile enthält ziemlich am Anfang eine `FROM`-Anweisung. Damit legt man das sogenannte Base-Image fest, in diesem Fall ist das ein Nginx-Server auf Basis der kompakten Linux-Distribution Alpine. Der Befehl `COPY` kopiert eine Datei aus dem Dateisystem, auf dem der Bauprozess läuft, ins Abbild. In diesem Fall landet die Datei index.htm, die neben dem Dockerfile liegt, im Verzeichnis /var/www/html. Wichtig für das Verständnis: Ist das Abbild gebaut, liegen alle Dateien darin. Die lokalen Dateien, die zum Bau gebraucht wurden, sind zum Betrieb des Abbilds nicht mehr nötig.

Ein weiterer häufig benutzter Befehl ist `RUN`. Damit kann man während des Bauprozesses Kommandozeilenbefehle innerhalb des Containers ausführen. Braucht man im Container etwa ein Paket vom Paketmanager, schreibt man im Dockerfile:

```
RUN apk add curl nano
```

Der Paketmanager von Alpine heißt `apk`. Der `RUN`-Befehl installiert die Pakete `curl` und `nano` – dadurch landen die beiden Programme im Dateisystem, das anschließend zum Container-Abbild verschnürt wird. `RUN` kann man für alle Arten von Befehlen nutzen, etwa um mit `mkdir` Verzeichnisse anzulegen oder Programmcode im Container zu kompilieren.

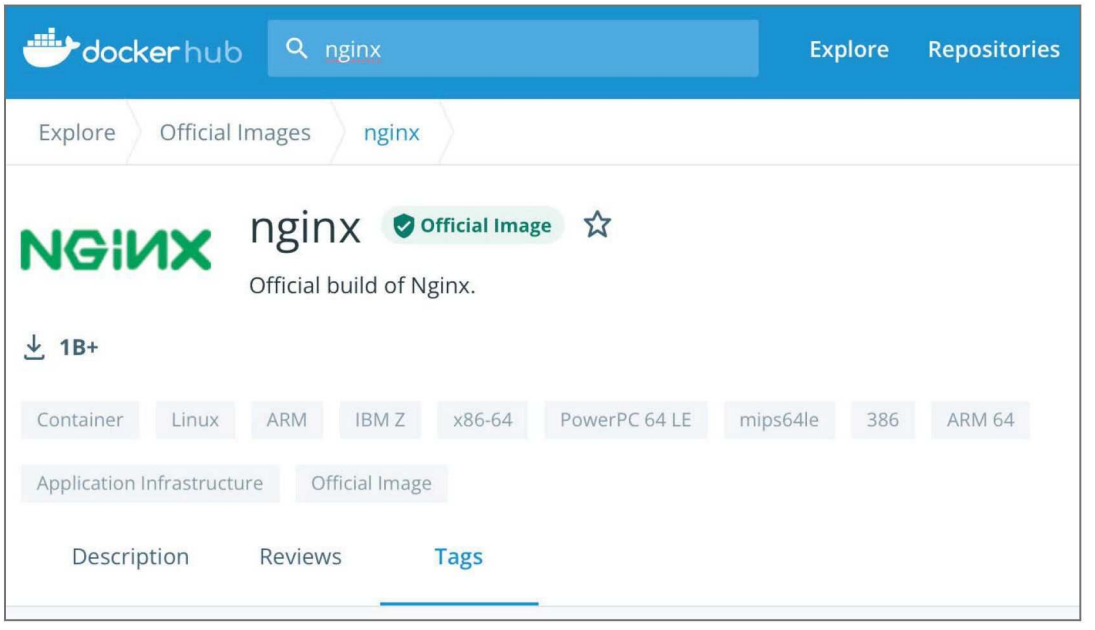
Ist das Dockerfile fertig geschrieben, startet man den Bauprozess auf der Kommandozeile mit dem folgenden Befehl:

```
docker build -t meinname/webserver .
```

Der Punkt am Ende weist Docker an, das Dockerfile im selben Ordner zu verwenden, der Parameter `-t` gibt dem Abbild den Namen `meinname/webserver`. Unter dem ist es jetzt im lokalen System erreichbar.

Abbilder verbreiten

Offen bleibt die Frage, wo im obigen Beispiel das Base-Image `nginx:alpine` herkommt. Hier kommt eine weitere Komponente ins Spiel, die die Firma Docker erfunden hat: die Container-Registry. Dockers



Images, die im Docker Hub mit „Official Image" markiert sind, werden von den Entwicklern der Software zusammen mit Docker-Mitarbeitern gepflegt und sind meist die beste Wahl.

Standard-Registry ist der Docker Hub, der auch eine Weboberfläche hat (hub.docker.com). Gute und geeignete Abbilder für eigene Projekte zu finden, ist eine Kunst, die man beim Dockern recht schnell lernen sollte. Grundsätzlich muss man dafür verinnerlichen: Der Docker Hub ist keine kuratierte Auswahl, sondern eine öffentliche Datenhalde für Abbilder. Jeder Docker-Nutzer mit einem Account für docker.com kann mit dem Befehl `docker push` ein lokal gebautes Abbild in den Docker Hub verschieben, für obiges Beispiel etwa mit

```
docker push meinname/webserver
```

Wenn man nicht aufpasst, stößt man daher schnell auf unsichere und schlechte Abbilder, oder gar solche, die in betrügerischer Absicht – zum Beispiel mit eingebautem Crypto-Miner – hochgeladen wurden.

Alle Abbilder liegen mit einer Bezeichnung nach dem Schema `benutzername/image-name:tag` im Docker Hub. Das Tag am Ende ist ein entscheidendes Detail: Tags kann man beim Hochladen an Abbilder hängen, um verschiedene Varianten zu kennzeichnen. Dabei gibt es keine verpflichtenden Regeln, nur Konventionen. Gängige Praxis ist es, Versionen einer Software nach dem Schema von Semantic Versioning zu taggen. Eine Software in Version 2.3.1 bekommt dann die Tags `:v2`, `:v2.3` und `:v2.3.1`. Die aktuelle Version bekommt zusätzlich das Tag `:latest`. Wer Container mit einem solchen Tag-Schema einsetzt, ist gut beraten, auf produktiven Systemen etwa `:v2.3` zu nutzen und jede neue Major-Version zuerst in einer Testumgebung auszuprobieren. Welche Tags es für ein Abbild gibt, erfahren Sie in der Weboberfläche des Docker Hubs.

Das Basis-Abbild aus dem Beispiel (`nginx:alpine`) hört gleich auf mehrere Tags: Zum Zeitpunkt, als der Artikel entstand, waren das: `1.21.3-alpine`, `mainline-alpine`, `1-alpine`, `1.21-alpine`, `alpine`. All diese Tags führen zum selben Ziel: Nginx in Version 1.21.3 mit einem Alpine-Unterbau.

Bei der Wahl des Unterbaus lohnt es, etwas genauer hinzusehen. Gibt es eine Alpine-Version, kann sich die durchaus lohnen, weil Alpine sehr kompakt ist und weniger Ballast mitschleppt, als etwa ein Debian Slim mitbringt. Am meisten Festplattenplatz und Downloadzeit spart man, wenn man auf einem Docker-Host möglichst wenige unterschiedliche Base-Images einsetzt.

Eine weitere gute Idee von Docker ist der Aufbau von Images: Sie bestehen aus Schichten – jede Zeile im Dockerfile setzt eine Schicht obendrauf. Im Hintergrund ist das ein weiterer Ordner mit einem Tar-Archiv. Jede Schicht enthält alle Abweichungen zur Schicht davor, also Hinzufügen und Löschen von Dateien und Ordnern. Alle Schichten werden beim Start des Containers übereinander gelegt und zu einem Dateisystem verknüpft. Doch die Magie geht noch weiter: Beim Download aus der Registry erkennt Docker anhand einer Prüfsumme, wenn der Daemon eine verlangte Schicht schon heruntergeladen hatte. Wenn Sie auf einem System also zehn verschiedene Images mit Alpine-Basis nutzen, wird die Alpine-Schicht nur beim ersten Mal heruntergeladen.

Vertrauenswürdige Abbilder

Weil jeder ohne Kontrolle alles Mögliche in den Hub schieben kann, sollten Sie von Images mit Namen wie `containerbastler76/nginx` die Finger lassen. Von vielen häufig genutzten Anwendungen finden Sie im Hub Abbilder ohne Benutzername und Schrägstrich davor, etwa `nginx` oder `mariadb`. Das sind sogenannte Official Images, die von den Maintainern der Software zusammen mit Entwicklern von Docker gepflegt werden. Sie sind in der Regel vertrauenswürdiger als Bastler-Images.

Neben den offiziellen Images finden Sie bei der Suche im Hub Abbilder mit einem blauen Haken und dem Hinweis „Verified Publisher“. Auch die sind ziemlich vertrauenswürdig – hinter dem Account stecken wirklich die Entwickler der jeweiligen Software, das hat die Firma Docker verifiziert.

Ein Problem, das Sie heute bei der Auswahl von Images nur noch selten lösen müssen, ist die Wahl der richtigen Prozessorarchitektur. Früher war es durchaus üblich, ein Image für ARM-Prozessoren (wie den im Raspberry Pi) mit einem Tag wie `:1-alpine-arm` zu markieren. Heute werden die meisten Images (vor allem die offiziellen) als sogenanntes Multi-arch-Image veröffentlicht. Docker lädt dann die passende Version für den verwendeten Prozessor herunter.

Installieren und einrichten

Mit diesem Wissen können Sie Ihren ersten Docker-Container, einen einfachen Nginx-Webserver, hochfahren. Dafür brauchen Sie nur noch Docker auf Ihrem System. Docker erscheint in zwei Varianten: Für Windows und macOS gibt es die Software Docker Desktop. Die ist zum Ausprobieren und Entwickeln gedacht, nicht für den Einsatz auf Servern. Docker Desktop enthält mehr als den reinen Docker Daemon, der Container ausführt. Obendrein enthält das Paket eine grafische Oberfläche für Einstellungen und eine Übersicht über laufende Container. Das ist alles ganz nettes Beiwerk, aber nicht essenziell – Docker bedient man meist auf der Kommandozeile.

Mit der Desktop-Version hat die Firma Docker jetzt ihre Nische im großen Cloud-Markt gefunden, mit der sie Geld verdienen wollen. Anstatt am Betrieb von Containern zu verdienen (den Markt haben sie lange an Kubernetes verloren), lassen sie sich jetzt von großen Firmen bezahlen, die Docker Desktop nutzen wollen: Wer Docker Desktop für Windows und macOS kommerziell nutzt und in einer Firma mit mehr als 250 Mitarbeitern oder mehr als 10 Millionen US-Dollar Jahresumsatz arbeitet, muss mindestens 5 US-Dollar im Monat zahlen. Die private oder kommerzielle Nutzung in kleinen Unternehmen bleibt kostenlos. Die Downloads für Windows und macOS (Intel- und Apple-Prozessor) finden Sie über ct.de/wzra.

Docker Desktop für macOS und Windows läuft mit einer Virtualisierungsebene: Auf dem Mac wird im Hintergrund eine einzige virtuelle Linux-Maschine für alle Container gestartet, von der man als Anwender aber nichts weiter mitbekommt. Wenn man viele Container gleichzeitig betreiben will, kann man ihr in der grafischen Oberfläche mehr Ressourcen zuweisen.

Auch unter Windows war eine VM lange die einzige Strategie – die virtuelle Maschine lief mit dem Windows-Hypervisor Hyper-V. Ab Windows 10 2004 (aus dem Frühjahr 2020) gibt es eine ressourcensparende Alternative: Docker läuft jetzt auch im neuen Windows Subsystem for Linux (WSL 2). Um diese Option im Docker-Installer direkt zu nutzen, sollten Sie vorab das WSL 2 unter Windows einrichten. Das geht auf einem neuen Windows 10 oder 11 sehr schnell mit einer Zeile auf der Kommandozeile oder in der PowerShell:

```
wsl --install
```

Anschließend kann man den grafischen Docker-Installer durchklicken und dort WSL 2 wählen.

Open Source für Linux

Für Linux – sowohl für Server als auch für Linux-Entwicklermaschinen mit grafischer Oberfläche – gibt

es den nackten Docker Daemon ohne die grafischen Extras der Docker-Desktop-Version. Dafür ist diese Software vollständig Open Source (sogenannte Community-Version) und darf auch von Unternehmen jeder Größe ganz ohne Abo und Account benutzt werden. Es spricht nichts dagegen, die Community-Version auf einem Server produktiv einzusetzen. Eine regelmäßig von uns aktualisierte Installationsanleitung für verschiedene Distributionen lesen Sie in einem kostenlosen Online-Artikel, zu finden über ct.de/wzra. Sollten Sie sich für die Details nicht interessieren, hier die Kürzestfassung.

Mit folgendem Befehl laden Sie das Installationsskript für Bequeme herunter:

```
curl -fsSL https://get.docker.com ↵
↳-o get-docker.sh
```

Das führen Sie aus und schauen Docker beim Installieren zu:

```
sudo sh get-docker.sh
```

Damit der aktuell angemeldete Nutzer mit Docker arbeiten darf, fügen Sie ihn zur Gruppe docker hinzu:

```
sudo usermod -aG docker $USER
```

Wenn Sie Docker für Ihr System installiert haben, öffnen Sie eine Kommandozeile und prüfen Sie mit `docker version`, dass die Installation erfolgreich war. Dann kann es mit dem Dockern losgehen.

Geld verdienen möchte Docker neuerdings auch mit dem Docker Hub und hat auch dort die Spielregeln geändert: Ohne Login kann man nur noch 100 Abbilder innerhalb von sechs Stunden herunterladen, um das Limit zu verdoppeln, braucht man einen Account (anzulegen über docker.com), mit dem man sich per `docker login` auf der Kommandozeile anmeldet. Wer Geld bezahlt, kann das Limit auf 5000 Abfragen pro Tag erhöhen. Jede Docker-Karriere beginnt so richtig mit dem Befehl `docker run`, der den ersten Container startet. Ein Nginx-Webserver ist ein gutes Beispiel zum Start. Führen Sie folgenden Befehl auf einer Maschine mit Docker aus:

```
docker run -p 8080:80 nginx:alpine
```

Erstes Erfolgserlebnis

Mit dem Parameter `-p` wird ein Port der Netzwerkkarten des Computers an einen Port im Container weitergegeben. Die Syntax `8080:80` bedeutet: Leite Anfragen, die an Port 8080 aller Netzwerkkarten ankommen, an Port 80 im Container weiter. Wenn Sie den Befehl ausführen, sehen Sie zuerst, wie das Abbild schichtweise aus dem Hub geladen wird. Dann vermeldet Nginx, dass es Worker-Prozesse gestartet hat. Öffnen Sie einen Browser und darin die Adresse http://localhost:8080 – Sie sollten die Nginx-Beispielseite sehen, die bereits Ihr erster Container ausliefert. Selbiges sollte (sofern keine Firewall im Weg ist) auch im internen Netzwerk mit der IP-Adresse Ihres Computers funktionieren, etwa http://192.168.178.105:8080.

Wenn sich Docker beschwert, dass der Port bereits belegt sei („port is already allocated“), ist das kein Unvermögen von Docker, sondern die grundlegende Funktionsweise von IP-Netzwerken: Nur ein Prozess darf gleichzeitig auf einem Port lauschen und irgendetwas auf Ihrem System hat diesen Port bereits okkupiert. Einen weiteren Container in einer weiteren Kommandozeilensitzung könnten Sie zum Beispiel auf Port 8081 starten: `-p 8081:80`. Wollen Sie nur eine Netzwerkkarte nutzen, setzen Sie deren IP-Adresse davor, zum Beispiel `-p 127.0.0.1:8081:80`.

Um laufende Container aufzulisten, öffnen Sie eine weitere Kommandozeilensitzung und führen dort `docker ps` aus. Diese Tabelle verrät Ihnen jederzeit, was auf dem Docker-Host gerade läuft. Jeder Container bekommt eine zufällig ausgewürfelte ID und einen kreativen Namen. Wenn Sie Ihren Containern sprechende Namen geben wollen, übergeben Sie `docker run` vor dem Namen des Images den Parameter `--name mein-webserver`. Mit dem Namen oder der ID, die Ihnen `docker ps` verraten, können Sie mit Containern einige Dinge anstellen: `docker stop <ID oder Name>` hält einen Container an. `docker rm <ID oder Name>` löscht ihn restlos. Hilfreich zum Fehlersuchen: `docker logs <ID oder Name>` zeigt die Logs eines Containers.

Diese Befehle sind das wichtigste Handwerkszeug im Docker-Alltag. Sobald Sie mehr als einen Container starten wollen, etwa um die oben erwähnte Zusammenstellung aus WordPress und Datenbank zu starten, wollen Sie aber nicht mehr mit `docker run` arbeiten.

Container im Verbund

Das Mittel der Wahl für Container-Zusammenstellungen heißt Docker Compose – und da läuft aktuell ein größerer Umbruch: Dieses Werkzeug wurde lange Zeit parallel zur Docker-CLI (in Go geschrieben)

in Python programmiert. Docker Compose war ein eigener Kommandozeilenbefehl namens `docker-compose` (mit Bindestrich). Das hat sich mittlerweile geändert und Docker Compose ist ein Subcommand des Befehls docker geworden, den man nicht mehr separat installieren muss.

Die Befehle `docker-compose` und `docker compose` verhalten sich fast identisch, sodass Sie alte Anleitungen einfach übernehmen und den Bindestrich weglassen können. Die Idee von Docker Compose: Statt Container einzeln zu starten, schreiben Sie YAML-Dateien, die mehrere Container (die man dort Service nennt) mit ihren Einstellungen beschreiben. Solche Compose-Dateien kann man irgendwo auf dem Desktop-PC oder Server anlegen. Am besten nutzt man für mehr Übersicht nur das Verzeichnis, in dem man auch all seine Softwareprojekte sammelt, und legt darin für jedes Docker-Projekt ein Verzeichnis an – bestenfalls versioniert man die Projekte direkt mit Git.

Was im obigen Beispiel noch in einem Kommandozeilenaufruf zusammengerührt war, steht in einer Docker-Compose-Datei, die man standardmäßig docker-compose.yml nennt, recht übersichtlich untereinander und nach YAML-Spielregeln eingerückt. Die Angabe der Version am Anfang ist Pflicht und alles andere als intuitiv – sie ändert sich ab und zu, wenn die Docker-Entwickler Änderungen an der Compose-Syntax vornehmen. Aktuell gibt es Version 3.8.

Um Docker Compose zu testen, stoppen Sie zuerst Ihre zuvor gestarteten Container mit `docker stop` und legen dann die Datei docker-compose.yml an:

```
version: "3.8"
services:
  mein-webserver:
    image: nginx:alpine
    restart: always
    ports:
     - 8080:80
```

Erzeugt werden soll ein Container mit dem Service-Namen `mein-webserver`. Port 8080 soll wie im Beispiel oben an Port 80 im Container ankommen. Weil die Ports eine YAML-Liste sind (mit - eingeleitet), können Sie hier beliebig viele Weiterleitungen für einen Container einrichten. Sehr nützlich ist für Server die Zeile `restart: always`. Sie weist den Docker Daemon an, den Container immer wieder zu starten – nach einem Neustart des Servers fährt also auch der Container wieder hoch.

Projekte für den Einstieg und Dokus

ct.de/wzra

Zum Start der Compose-Zusammenstellung navigieren Sie auf der Kommandozeile in den Ordner mit der YAML-Datei und setzen dann den Befehl zum Start ab:

```
docker compose up -d
```

Der Parameter `-d` aktiviert den Detached-Modus, die Log-Ausgaben blockieren so nicht die aktuelle Kommandozeilensitzung und der Container läuft im Hintergrund. Ein Besuch von http://localhost:8080 im Browser sollte wieder die Nginx-Seite zeigen.

Das Gegenstück zu `docker compose up` ist `docker compose down`. Wichtig zum Verständnis: Docker Compose arbeitet immer mit der Datei docker-compose.yml im aktuellen Verzeichnis. Wenn Sie mehrere Zusammenstellungen auf einer Maschine haben, können Sie die separat hoch- und runterfahren, indem Sie das Verzeichnis wechseln und dort `docker compose` ausführen.

Lernpfad

Jetzt sind Sie bereit für die ersten nützlichen Docker-Projekte. Am schnellsten lernt man Vorteile und Tücken von Containern, wenn man sich einen Raspi oder eine Entwicklermaschine schnappt, dort fertige Compose-Projekte ausprobiert und ein paar Stellschrauben verändert. Über ct.de/wzra finden Sie von uns in der Vergangenheit vorgestellte Projekte, die sich für den Start gut eignen, unter anderem eine WordPress-Zusammenstellung, Nextcloud und einen Passwort-Tresor. Dabei begegnen Ihnen zwei weitere Konzepte: Mit Volumes speichern Sie Daten dauerhaft, sodass sie das Entfernen eines Containers überstehen. Mit Umgebungsvariablen steuern Sie die Funktion des Containers.

Wenn Sie sich sicher mit fertigen Abbildern und Zusammenstellungen fühlen, können Sie sich ans Schreiben eigener Dockerfiles für eigene Software wagen und diese mit `docker push` im Docker Hub veröffentlichen. Inspiration für gute Dockerfiles gibt es jeweils in GitHub-Repositories großer Projekte, die mit derselben Programmiersprache arbeiten. Organisationen, die künftig auf Container setzen wollen, müssen sich dann eine Strategie überlegen, wie sie ihre Images bereitstellen. Anregungen finden Sie im nachfolgenden Artikel. Wenn Sie bisher mit dem Dockern gewartet haben, sind Sie jetzt im Vorteil: Die meisten schwerwiegenden Probleme haben in den vergangenen zehn Jahren schon andere für Sie gelöst. (jam) c't

Die Container-Strategie

Container kapseln Prozesse und lösen Probleme mit Abhängigkeiten, so die Theorie. Doch damit Containerisierung einer Organisation echten Nutzen bringt, braucht diese eine Container-Strategie: Wo lagern die Images, wer baut sie und wie werden sie aktualisiert? Anregungen aus der Praxis und Überlegungen zur Sicherheit.

Von **Jan Mahn**

Der Einstieg in die Welt der Containerisierung ist einfach. Mit Docker oder Podman fahren Entwickler und Admins nach kurzer Einarbeitungszeit in Containern Anwendungen hoch, deren Installation mit konventionellen Werkzeugen Stunden dauern kann. Ebenso schnell hat man die Anwendung in einer Docker-Compose- oder Podman-Compose-Datei definiert und auf Entwicklermaschine oder Produktivsystem hochgefahren. Und selbst der Umstieg von Compose-YAML auf Kubernetes ist kein Hexenwerk, wie die Einführung „Der Lernpfad zum Kubernetes-Kenner“ ab Seite 38 beweist.

Solange man plant, nur Anwendungen in Containern zu betreiben, die andere bereits verpackt haben, ist die Arbeit vergleichsweise schnell erledigt. Ein oder Hundert WordPress-Blogs? Schnell gestartet, weil WordPress unter dem Namen `wordpress` fertig verpackt im Docker-Hub liegt. Ebenso eine passende Datenbank (`mariadb` oder `mysql`). Gleiches gilt für eine selbstgehostete Nextcloud-Instanz (`nextcloud`).

Wohin mit den Images

Ungleich komplizierter wird es, sobald man selbst Software entwickelt und in Containern bereitstellen will – für den Eigenbedarf, für Kunden oder für die Open-Source-Gemeinschaft. Dann braucht die Organisation unweigerlich eine Ablage für Images. In den Anfangstagen der Containerei war der Docker-Hub die erste Wahl. Open-Source-Images lagerten dort grundsätzlich kostenlos, Firmen konnten private Abbilder dort ablegen. Doch im Frühjahr 2023 verursachte die Firma Docker Inc., Betreiberin des Docker-Hubs, ein mittelschweres Beben. In einem Blogpost kündigte man an, dass Open-Source-Projekte nicht länger automatisch einen kostenlosen Team-Account bekommen und sich stattdessen für das „Docker-Sponsored Open Source Program“ bewerben müssen. Wer das nicht rechtzeitig versuche, müsse damit rechnen, dass seine Abbilder gelöscht werden. Wenig später folgten zwei weitere Blogposts, in denen man sich für die Kommunikation entschuldigte und die Ankündigungen zurücknahm (siehe ct.de/wr6h). Doch für viele Projekte kam die Entschuldigung zu spät, die Suche nach alternativen Container-Registries hatte bereits begonnen.

Und auch für Unternehmen kann sich ein Blick über den Tellerrand lohnen, denn der Docker-Hub ist nicht die einzige Registry. Wer heute ein Container-Image anbieten will, hat die Wahl zwischen mehreren Registries, die für öffentliche Images kostenlos und für private kostenpflichtig sind. Red Hat betreibt quay.io, wie Podman besonders im Red-Hat- und Fedora-Umfeld populär. Registries gibt es ferner auch bei den Cloudprovidern wie Google Cloud, AWS und Microsoft. Infrage kommen sie vor allem dann, wenn die Anwendungen in den Rechenzentren dieser Anbieter laufen sollen.

Wer seinen Code bei GitHub lagert, kann seine Images in GitHubs eigener Registry (ghcr.io) ablegen – im Zusammenspiel mit der CI/CD-Umgebung (Continuous Integration/Continuous Delivery) GitHub Actions landen neue Images dort ganz automatisch. Ein vergleichbares Angebot gibt es bei GitLab mit registry.gitlab.com. Egal in welcher Registry man seine Abbilder lagert: CI/CD gehört in jedem Fall zu einer Container-Strategie. Rein technisch kann jeder Entwickler mit `docker push` lokal gebaute Images von seiner Entwicklermaschine in eine Registry schubsen, sofern man ihm die Rechte dafür eingeräumt hat. Guter Stil ist das aber nicht: Denn Code im Allgemeinen und Images im Speziellen sollten nur dann auf einer produktiven Maschine im eigenen Haus oder beim Kunden landen, wenn sie von automatisierten Tests geprüft und in einer reproduzierbaren Umgebung gebaut wurden. Das kann nur eine CI/CD-Umgebung leisten. Nur dann ist sichergestellt, dass man zu 100 Prozent sicherstellen

kann, welcher Code und welche Konfiguration in einem Image stecken.

Eigene Registry

Die Alternative zu einer Registry, die andere im Internet bereitstellen, ist eine selbstbetriebene Registry. Auch dafür kommen wieder GitHub (in der selbstgehosteten Enterprise-Variante) und GitLab infrage. Oder die Open-Source-Registry Harbor, ein Projekt unter dem Dach der Cloud Native Computing Foundation (CNCF). Die steckt selbst in Containern und kann im eigenen Netzwerk oder im Internet veröffentlicht werden. Im Funktionsumfang ist eine Weboberfläche enthalten, über die man seine Abbilder verwalten kann. Für große Organisationen gibt es eine ausgefeilte Benutzer- und Berechtigungsverwaltung, Einblicke in die Abhängigkeiten sowie eine Integration des Image-Scanners Trivy, dessen Funktionsweise wir im Artikel „Container-Images mit Trivy durchleuchten“ ab Seite 18 vorstellen. Für ein Dreipersonenunternehmen lohnt der Betrieb einer eigenen Harbor-Instanz eher nicht, dafür erfüllt es viele Wünsche von großen Organisationen. Nicht abschrecken lassen sollten sich Docker- und Podman-Nutzer von der Homepage des Harbor-Projekts (goharbor.io). Dort wird die Software als ideale Registry für Kubernetes vorgestellt – doch im Kern handelt es sich um eine Registry, die dem Standard der OCI (Open Container Initiative) folgt und mit Docker und Podman gleichermaßen harmoniert.

Der Clou an einer selbstgehosteten Registry: Sie kann nicht nur selbstgebaute Images verwalten, man kann sie auch dazu bringen, Abbilder aus einer öffentlichen Registry im Internet herunterzuladen, aktuell zu halten und intern anzubieten. Damit kann man Sicherheitsprobleme einschränken, die aus leichtsinnigem Gebrauch von öffentlichen Images resultieren. Entwicklern wird es untersagt, Images aus dem Docker-Hub und anderen Registries zu laden.

Öffentlich und privat

Hat man sich für eine Registry entschieden und einen Plan, wie der eigene Code verpackt und dort abgelegt wird, müssen Entwicklerrechner und Server berechtigt werden, sofern die Images nicht öffentlich sind. Docker kennt zum Anmelden den Befehl `docker login`, bei Podman heißt das Gegenstück `podman login`. Der Dokumentation der jeweiligen Registry sollte man unbedingt entnehmen, wie man Anmeldetoken erzeugt. Anders als das Benutzerkennwort kann man mit einem solchen Token nur explizit ausgewählte Funktionen ausführen. Ein Docker-Server zum Beispiel braucht nur ein Token, der Container aus der Registry abholen kann – und nicht das Recht, selbst Images zu schreiben.

In Kubernetes ist es nicht ganz so leicht, sich an einer privaten Registry anzumelden. Der schnellste Weg führt über einen PC mit installiertem Docker. Hier meldet man sich mittels `docker login` und einem Token an der Registry an. Docker erzeugt eine Datei namens config.json und legt sie im lokalen Benutzerprofil unter dem Pfad ~/.docker/config.json ab. Darin liegen auch die Zugangsdaten, die Datei muss also entsprechend behandelt werden. Das Kubernetes-CLI Kubectl kennt einen Befehl, um daraus ein Secret zu machen, das im Cluster liegt und dort zum Download von privaten Images genutzt werden kann. Das folgende erzeugt ein Secret namens `mycred` aus der Docker-Datei:

```
kubectl create secret generic mycred \
--from-file=.dockerconfigjson=~/.docker/config.
json> \
--type=kubernetes.io/dockerconfigjson
```

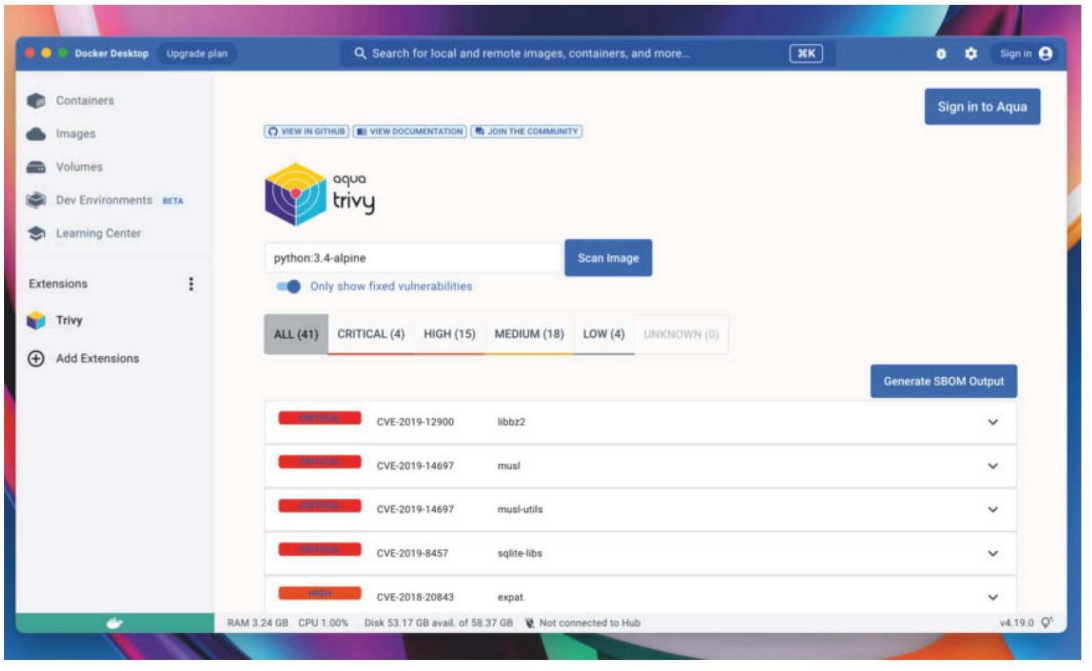
Beim Anlegen eines Pods kann man dieses Secret als imagePullSecret einsetzen:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-private-nginx
spec:
  containers:
  - name: nginx
    image: registry.example.org/my/nginx:latest
  imagePullSecrets:
  - name: mycred
```

Fazit

Mit dem Einrichten von Docker, Kubernetes oder Podman allein ist es nicht getan. Zum Containerbetrieb im professionellen Umfeld gehören mindestens eine Registry für eigene Images und bestenfalls auch eine CI/CD-Umgebung. Viele der Entscheidungen auf dem Weg betreffen direkt die Sicherheit und sollten nicht leichtfertig getroffen werden. Die gute Nachricht: Wenn Sie jetzt eine Container-Strategie für Ihre Organisation planen, haben Sie die Chance, Sicherheitsüberlegungen von Anfang an einfließen zu lassen. (jam) c't

Dokumentationen
ct.de/wr6h



Container-Images mit Trivy durchleuchten

Docker verführt wegen der einfachen Handhabung dazu, Container-Images als Blackbox zu behandeln. Welche Abhängigkeiten in einem Image stecken und ob diese Sicherheitslücken mitbringen, ist nicht immer klar ersichtlich. Mit dem Open-Source-Scanner Trivy klopfen Sie Ihre Images auf Schwachstellen ab.

Von **Niklas Dierking**

Container erfreuen sich bei Entwicklern, Administratoren und Anwendern großer Beliebtheit, weil sie viel Arbeit abnehmen. Alle Abhängigkeiten, die ein laufender Prozess im Container benötigt, stecken bereits im Container-Image. Einen einsteigerfreundlichen Überblick über Docker verschafft Ihnen der Artikel „Container verstehen und loslegen“ ab Seite 8.

Der Docker Hub ist die größte Sammlung von Container-Images (eine sogenannte Container-Registry). Dort kann jeder Images für alle möglichen Anwendungsbereiche hochladen. Ob ein Image vertrauenswürdig ist oder Sicherheitslücken enthält, ist jedoch nicht leicht auf Anhieb zu erkennen.

Ein Forscherteam der Technisch-Naturwissenschaftlichen Universität Norwegen untersuchte im

Jahr 2020 insgesamt 2500 zufällig ausgewählte Docker-Images aus dem Docker Hub auf bekannte Sicherheitslücken. Von den zertifizierten Images wiesen 82 Prozent mindestens eine Sicherheitslücke auf, die laut Bewertungssystem CVSS (Common Vulnerability Scoring System) als „schwerwiegend“ gelten. Bei den Community-Images waren es 68 Prozent der Abbilder. Am besten haben die offiziellen Images abgeschnitten, die das Unternehmen Docker selbst auf Schwachstellen kontrolliert. Aber auch diese enthielten noch zu 46 Prozent eine schwerwiegende Lücke. Die Studie haben wir unter ct.de/wq23 verlinkt.

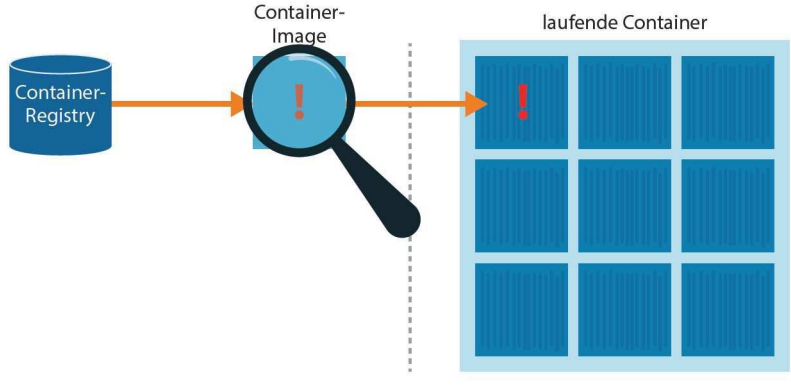
Mit dem Open-Source-Tool „Trivy“ durchleuchten Sie Images nach bereits katalogisierten Sicherheitslücken. Trivy gleicht dazu die Betriebssystempakete des Basis-Image (Alpine, Debian, Ubuntu), sprachenspezifische Pakete (npm, yarn, cargo), Bibliotheken und weitere Abhängigkeiten mit einem eigenen Schwachstellenregister ab. Das befüllt Trivy aus der etablierten Schwachstellendatenbank NVD (National Vulnerability Database), die sich aus dem CVE-Verzeichnis (Common Vulnerabilities and Exposures System) speist und Sicherheitslücken mit dem Bewertungssystem CVSS (Common Vulnerability Scoring System) einen Schweregrad zuweist. Darüber hinaus nutzt Trivy die CVE-Tracker und Security-Ratgeber diverser Open-Source-Projekte wie den Ubuntu-CVE-Tracker, die Rustsec-Advisories oder die Go Vulnerability Database.

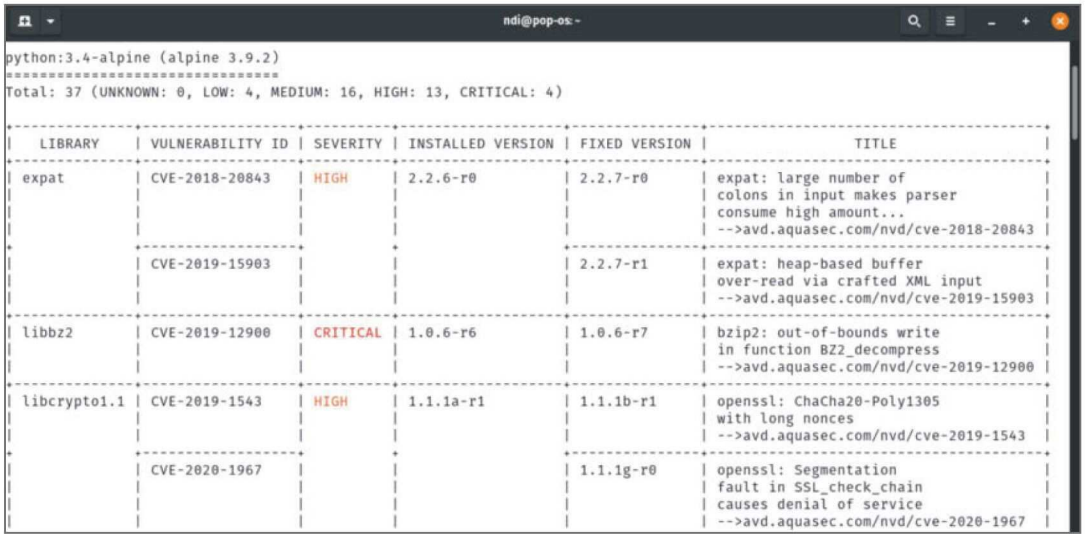
Images durchleuchten

Sie können Trivy einsetzen, um lokale Container-Images zu scannen, ebenso wie entfernte Abbilder, die in einer Container-Registry wie dem Docker Hub oder der GitHub-Container-Registry abgelegt sind. Das funktioniert auch mit Images, die Sie mit dem Befehl `docker save` als Tar-Archiv exportiert haben. Die Software unterstützt das von Docker verwendete OCI- und das Podman-Format. Ebenso durchsucht Trivy den Code öffentlicher GitHub-Repositories nach bekannten Schwachstellen. Trivy gibt es als Paket für die Linux-Distributionen RHEL, CentOS, Debian, Ubuntu und Arch-Linux sowie für macOS via Homebrew-Paketmanager. Eine Anleitung, wie Sie Trivy auf Ihrem System installieren, finden Sie in der Dokumentation des Projekts, die wir unter ct.de/wq23

Container-Scanner

Trivy sucht lokale und entfernte Container-Images nach bekannten Sicherheitslücken ab. Dabei prüft Trivy sowohl Betriebssystempakete (Alpine, Debian, Ubuntu) als auch Programmiersprachen-spezifische Pakete (yarn, npm, cargo). Das hilft Nutzern bei der Risikoeinschätzung.





```
ndi@pop-os: ~
python:3.4-alpine (alpine 3.9.2)
================================
Total: 37 (UNKNOWN: 0, LOW: 4, MEDIUM: 16, HIGH: 13, CRITICAL: 4)

+---------------+------------------+----------+-------------------+---------------+---------------------------------------+
|    LIBRARY    | VULNERABILITY ID | SEVERITY | INSTALLED VERSION | FIXED VERSION |                 TITLE                 |
+---------------+------------------+----------+-------------------+---------------+---------------------------------------+
| expat         | CVE-2018-20843   | HIGH     | 2.2.6-r0          | 2.2.7-r0      | expat: large number of                |
|               |                  |          |                   |               | colons in input makes parser          |
|               |                  |          |                   |               | consume high amount...                |
|               |                  |          |                   |               | -->avd.aquasec.com/nvd/cve-2018-20843 |
|               +------------------+          |                   +---------------+---------------------------------------+
|               | CVE-2019-15903   |          |                   | 2.2.7-r1      | expat: heap-based buffer              |
|               |                  |          |                   |               | over-read via crafted XML input       |
|               |                  |          |                   |               | -->avd.aquasec.com/nvd/cve-2019-15903 |
+---------------+------------------+----------+-------------------+---------------+---------------------------------------+
| libbz2        | CVE-2019-12900   | CRITICAL | 1.0.6-r6          | 1.0.6-r7      | bzip2: out-of-bounds write            |
|               |                  |          |                   |               | in function BZ2_decompress            |
|               |                  |          |                   |               | -->avd.aquasec.com/nvd/cve-2019-12900 |
+---------------+------------------+----------+-------------------+---------------+---------------------------------------+
| libcrypto1.1  | CVE-2019-1543    | HIGH     | 1.1.1a-r1         | 1.1.1b-r1     | openssl: ChaCha20-Poly1305            |
|               |                  |          |                   |               | with long nonces                      |
|               |                  |          |                   |               | -->avd.aquasec.com/nvd/cve-2019-1543  |
|               +------------------+          |                   +---------------+---------------------------------------+
|               | CVE-2020-1967    |          |                   | 1.1.1g-r0     | openssl: Segmentation                 |
|               |                  |          |                   |               | fault in SSL_check_chain              |
|               |                  |          |                   |               | causes denial of service              |
|               |                  |          |                   |               | -->avd.aquasec.com/nvd/cve-2020-1967  |
```

Trivy listet alle relevanten Informationen zu gefundenen Schwachstellen in einer übersichtlichen Kommandozeilenausgabe auf. CVE-Nummer und Kurzbeschreibung dienen der weiteren Recherche.

python:3.4-alpine (alpine 3.9.2) - Trivy Report - 2021-10-20T08:22:26.376586799Z

alpine

Package	Vulnerability ID	Severity	Installed Version	Fixed Version	Links
expat	CVE-2018-20843	HIGH	2.2.6-r0	2.2.7-r0	http://lists.opensuse.org/opensuse-security-announce/2019-07/msg00039.html https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=5226 https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=931031 Toggle more links
expat	CVE-2019-15903	HIGH	2.2.6-r0	2.2.7-r1	http://lists.opensuse.org/opensuse-security-announce/2019-09/msg00080.html http://lists.opensuse.org/opensuse-security-announce/2019-09/msg00081.html http://lists.opensuse.org/opensuse-security-announce/2019-11/msg00000.html Toggle more links
libbz2	CVE-2019-12900	CRITICAL	1.0.6-r6	1.0.6-r7	http://lists.opensuse.org/opensuse-security-announce/2019-07/msg00040.html http://lists.opensuse.org/opensuse-security-announce/2019-08/msg00050.html http://lists.opensuse.org/opensuse-security-announce/2019-11/msg00078.html Toggle more links
libcrypto1.1	CVE-2019-1543	HIGH	1.1.1a-r1	1.1.1b-r1	http://lists.opensuse.org/opensuse-security-announce/2019-07/msg00056.html https://access.redhat.com/errata/RHSA-2019:3700 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-1543 Toggle more links
libcrypto1.1	CVE-2020-1967	HIGH	1.1.1a-r1	1.1.1g-r0	http://lists.opensuse.org/opensuse-security-announce/2020-07/msg00004.html http://lists.opensuse.org/opensuse-security-announce/2020-07/msg00011.html http://packetstormsecurity.com/files/157527/OpenSSL-signature_algorithms_cert-Denial-Of-Service.html Toggle more links
libcrypto1.1	CVE-2021-23840	HIGH	1.1.1a-r1	1.1.1j-r0	https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-23840 https://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=6a51b9e1d0cf0bf8515f7201b68fb0a3482b3dc1 https://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=9b1129239f3ebb1d1c98ce9ed41d5c9476c47cb2 Toggle more links

Trivy beherrscht unterschiedliche Ausgabe-Formate, beispielsweise kann es eine HTML-Datei generieren, die Ihnen zur weiteren Verarbeitung der Ergebnisse dient.

verlinkt haben. Wenn Sie mit Docker Desktop oder Podman Desktop containern, können Sie auch die integrierte Trivy-Extension benutzen.

Wenn Sie Trivy auf dem Hostsystem installiert haben, dann scannen Sie ein Image, beispielsweise Python 3.4 mit Alpine-Linux als Basis, mit dem folgenden Befehl:

```
trivy image python:3.4-alpine
```

Außer Container-Images können Sie auch öffentliche GitHub-Repositories nach Schwachstellen durchsuchen:

```
trivy repo ↵
↳https://github.com/knqyf263/trivy-ci-test
```

Wenn Sie Trivy installiert haben, dann lädt das Tool beim ersten Scan die Schwachstellendatenbank

herunter, was einige Sekunden in Anspruch nehmen kann. Die nachfolgenden Scans erfolgen deutlich schneller, weil Trivy die Datenbank lokal als Cache ablegt und künftig nur noch Änderungen herunterlädt. Nutzen Sie Trivy selbst als Docker-Container, dann wird die Datenbank nicht automatisch zwischengespeichert, es sei denn, Sie reichen einen Ordner für den Cache in den Container hinein. Das erledigen Sie, indem Sie den `docker run`-Befehl wie folgt anpassen:

```
docker run --rm -v ${PWD}/cache:↵
↳/root/.cache/ aquasec/trivy ↵
↳python:3.4-alpine
```

Nach dem Abgleich mit der Schwachstellendatenbank präsentiert Trivy Ihnen die Ergebnisse des Scans standardmäßig in einer Tabelle auf der Kommandozeile.

In der Kopfzeile der Tabelle finden Sie eine Zusammenfassung der Ergebnisse. Für das Beispiel-Image `python:3.4-alpine` hat Trivy insgesamt 37 Schwachstellen gefunden, denen es die Schweregrade „Unknown“, „Low“, „Medium“, „High“ und „Critical“ in der Spalte „Severity“ zuordnet. In der Tabelle finden Sie unter „Library“ auf der linken Seite das Paket oder die Programmbibliothek, die die Schwachstelle enthält. „Vulnerability-ID“ listet die Kennung, mit der die Schwachstelle katalogisiert wurde. Der Spalte „Installed Version“ entnehmen Sie, in welcher Version die betroffene Abhängigkeit vorliegt. Besonders praktisch: Unter „Fixed Version“ informiert Trivy Sie direkt, in welcher Version die Entwickler das Problem behoben haben. In der „Title“-Spalte ganz rechts lesen Sie eine Kurzbeschreibung der Schwachstelle und finden einen Link mit weiterführenden Informationen.

Blick schärfen

Lassen Sie sich von der Flut der gefundenen Schwachstellen zunächst nicht verunsichern. Nur weil ein Container-Image bekannte Schwachstellen enthält, heißt das nicht automatisch, dass in Ihrem konkreten Anwendungsfall ein Sicherheitsrisiko besteht. So könnte es möglich sein, dass der Prozess im laufenden Container gar nicht mit der betroffenen Abhängigkeit interagiert. Sie müssen also den Einzelfall prüfen.

Um die Analyse einzugrenzen, können Sie Trivy auch anweisen, nur besonders schwerwiegende Schwachstellen auszugeben. Mit folgendem Befehl lassen Sie sich beispielsweise nur Lücken anzeigen, die als „High“ oder „Critical“ eingestuft werden:

```
trivy image --severity HIGH,CRITICAL \
python:3.4-alpine
```

Wenn Sie bereits recherchiert haben, dass bestimmte Lücken für Ihren Anwendungsfall nicht relevant sind, dann können Sie in dem Verzeichnis, in dem Sie Trivy ausführen, auch eine Textdatei namens `.trivyignore` anlegen. Dort gelistete Schwachstellen ignoriert Trivy beim nächsten Scan. Beispiel:

```
# Risiko ist akzeptabel
CVE-2019-19242
# Für Anwendungsfall nicht relevant
CVE-2020-11655
```

Seine volle Stärke entfaltet Trivy, wenn Sie Scans automatisieren, beispielsweise mittels GitHub Actions. So können Sie beispielsweise spezifizieren, welchen Exit-Code Trivy ausgeben soll, wenn bestimmte Schwachstellen gefunden werden:

```
trivy image --exit-code 0 --severity \
MEDIUM,HIGH python:3.4-alpine

trivy image --exit-code 1 --severity \
CRITICAL python:3.4-alpine
```

Das ist besonders hilfreich, wenn Sie planen, Trivy in automatisierte Buildprozesse oder eine CI/CD-Lösung einzubinden. In diesem Beispiel schlägt der Buildprozess fehl, wenn Trivy eine kritische Sicherheitslücke aufspürt und den Exit-Code 1 ausgibt. Der Prozess läuft aber durch, falls es sich lediglich um eine mittelschwere Schwachstelle handelt. Die Entwickler stellen in der Dokumentation des Projektes Vorlagen bereit (siehe ct.de/wq23), um Trivy in GitHub Actions und weitere CI/CD-Lösungen wie GitLab CI oder Travis CI zu integrieren.

Fazit

Mit Trivy verschaffen Sie sich im Handumdrehen einen Überblick über mögliche Schwachstellen in Container-Images und können so eine Risikoeinschätzung vornehmen. Das Scan-Tool hilft Administratoren und Container-Enthusiasten bei der Suche nach Images mit der kleinstmöglichen Angriffsfläche. DevOps unterstützt Trivy als Kontrollmechanismus in automatisierten Builds. (ndi) c't

Trivy-Dokumentation und GitHub-Repository

ct.de/wq23

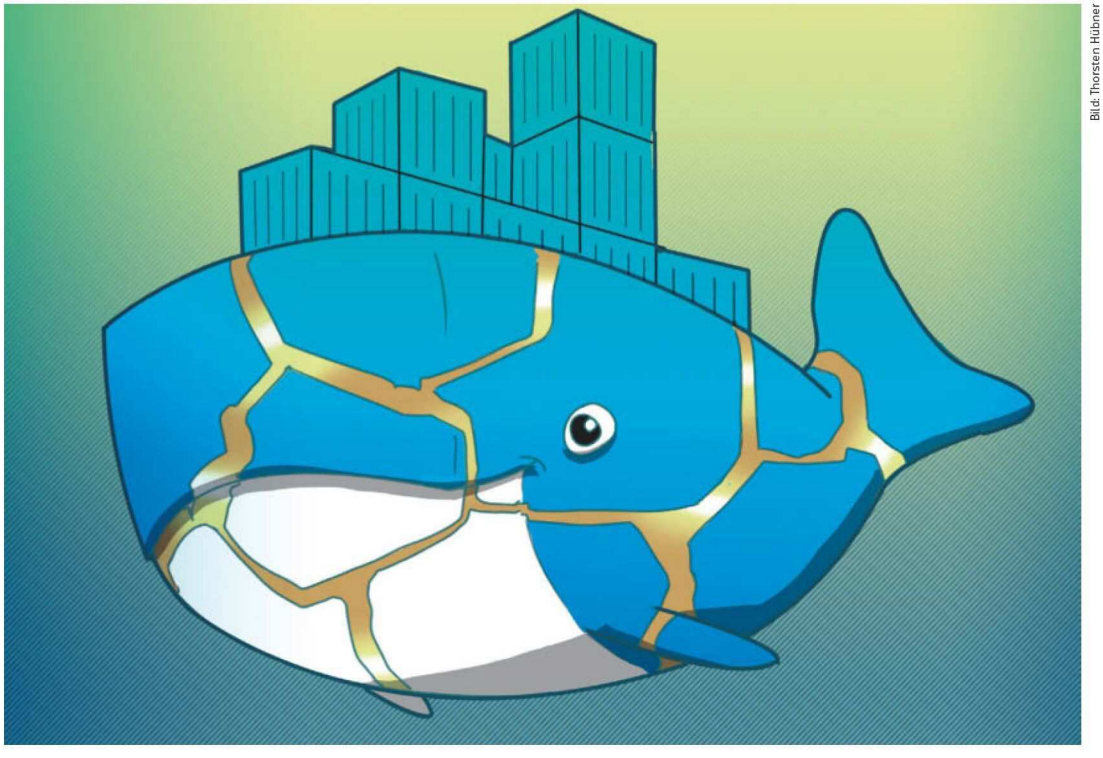


Bild: Thorsten Hübner

So harmonieren Docker und IPv6

Globale IPv4-Adressen sind knapp. Deshalb sollten heute im Netzwerk angebotene Dienste selbstverständlich per IPv6 erreichbar sein. Doch die verbreitete Container-Umgebung Docker erschwert es, IPv6 sinnvoll einzusetzen. Unser Artikel sortiert die Einzelteile, um sie besser zusammenzufügen.

Von **Peter Siering**

Dass Docker und IPv6 fremdeln, wird an vielen Stellen deutlich. Sichtbar zu Tage tritt es, wenn man die Log-Daten von Containern studiert: Während dort die IPv4-Adressen der anfragenden Clients auftauchen, findet sich für Anfragen von IPv6-Clients darin nur die IPv4-Adresse der internen Netzwerkschnittstelle „docker0“. Um dieses Problem zu lösen und für weitere präpariert zu sein, hilft es, die Docker-Netzwerkmöglichkeiten zu rekapitulieren.

Docker kennt grundsätzlich mehrere Techniken, um Container mit dem lokalen Netz zu verbinden. Dieser Artikel betrachtet den meistgenutzten Typ „Bridge“ und den produktiven Betrieb auf einem Linux-Server. Hier helfen spezielle Mechanismen, um die Container vom Netzwerk des Hosts zu separieren, um ihnen untereinander die Kommunikation zu erlauben und um Zugriffe von außen auf Dienste in den Containern zu realisieren.

Andere Netzwerktypen kommen bei besonderen Wünschen zum Einsatz: der Typ „Overlay“, wenn Container über mehrere Hosts verteilt erreichbar sein sollen, die Typen „Host“ und „macvlan“, wenn ein Container direkt am Netzwerk des Hosts lauschen soll, etwa um dort Broadcasts zu empfangen oder zu senden, und der Typ „ipvlan“, um komplexe virtuelle Netzwerke zu bauen. Alle Typen funktionieren grundsätzlich sowohl mit IPv4 als auch mit IPv6.

Bockige Brücke

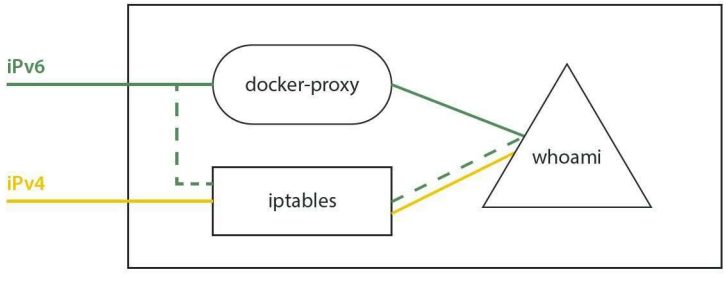
Für die meisten Anwendungsfälle genügt ein Bridge-Netzwerk. In einer regulären Installation kümmert sich Docker auf Anforderung darum, dass im Container laufende Netzwerkdienste von außen über die IP-Adresse des Docker-Hosts erreichbar sein sollen. Das kann man sich wie eine Portfreigabe beim DSL-Router vorstellen. Das Image deklariert dazu, welche Ports es exponiert, beim Starten des Containers weist Docker ihnen Ports des Hosts zu, einem simplen Webserver etwa Port 80.

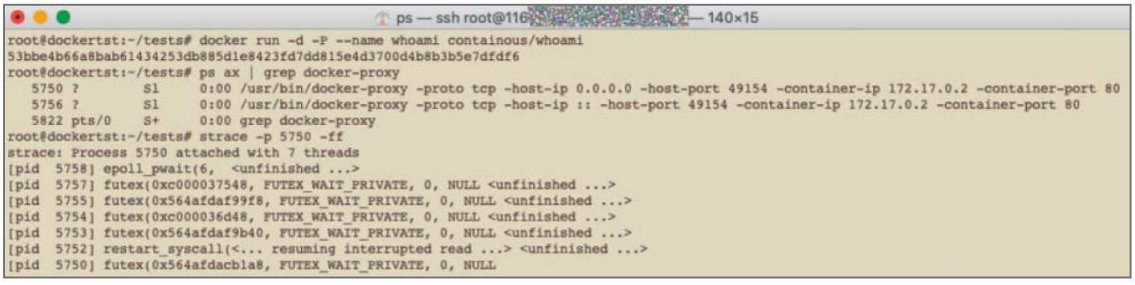
Auf dem Docker-Host auf Port 80 eingehende Netzwerkzugriffe leitet der dann an den Container weiter. Der Host kann dabei weltweit erreichbare globale IP-Adressen besitzen, etwa als Mietserver eines Cloud-Hosters. Der Container hat standardmäßig nur eine private IPv4-Adresse, die Docker dem gängigen für die lokale Nutzung reservierten Adressraum entnimmt (etwa 172.18.0.0/16).

Ob die externen Zugriffe auf IPv4- oder IPv6-Adressen des Hosts erfolgen, spielt keine Rolle. Es lohnt aber ein genauer Blick hinter die Kulissen: Auf einer Linux-basierten Installation sowohl mit IPv4- als auch mit IPv6-Adresse startet Docker für jedes Protokoll und jeden exponierten Port einen eigenen Prozess namens docker-proxy, der Zugriffe auf den IPv4- oder IPv6-Port des Hosts an den IPv4-Port des Containers weiterreicht. Pro Port laufen

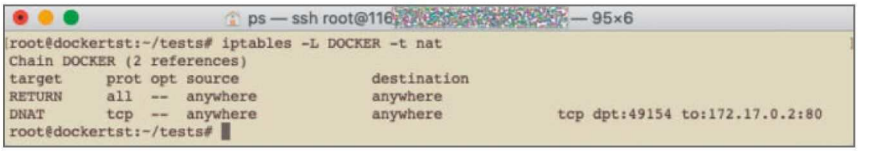
Docker IPv4 versus IPv6

Die Standardinstallation von Docker auf Linux-Hosts sieht unterschiedliche Wege für Netzwerkzugriffe auf Container in einem Bridge-Netzwerk vor. IPv4-Zugriffe (gelb) landen über von Docker eingerichtete Firewallregeln im Container, IPv6-Zugriffe (grün) hingegen fließen über einen speziellen docker-proxy-Prozess. Erst nach dem Aktivieren experimenteller Funktionen verwendet Docker für IPv6 ebenfalls Firewallregeln (grün gepunktet).





Wenige Befehlszeilen helfen dabei zu überprüfen, auf welchem Weg per IPv4 und IPv6 eingehende Anfragen bei einem Container landen.



Die bei IPv4 standardmäßig aktiven Firewallregeln für DNAT zeigt ein Aufruf von `iptables` an.

also auf einem Dual-Stack-Host zwei der docker-proxy-Prozesse.

Auf Linux-Systemen richten aktuelle Docker-Versionen zusätzlich Firewallregeln ein, die den gleichen Zweck erfüllen wie die docker-proxy-Prozesse – allerdings nur für IPv4. Das heißt, dass bei aktuellen Installationen IPv4-Zugriffe allein über Firewall-Regeln in den Container gelangen, IPv6-Zugriffe nur über die docker-proxy-Prozesse.

Spurensuche

Das lässt sich recht leicht überprüfen: Lassen Sie einen einfachen Container wie „containous/whoami“ laufen, der einen minimalen Webserver enthält und die HTTP-Request-Daten zurückliefert (alle erwähnten Container-Images unter ct.de/wp8n). Wenn Sie von einem Client aus per IPv4 zugreifen, sehen Sie in der von containous/whoami erzeugten Ausgabe die IP-Adresse des Clients als „RemoteAddr“. Wiederholen Sie den Zugriff auf die IPv6-Adresse, taucht in „RemoteAddr“ die IP-Adresse auf, die auf dem Docker-Host für das docker0-Interface konfiguriert ist (das Default-Bridge-Netz).

Sie können sich mit dem Linux-Prozessmonitor `strace` vergewissern, dass der für IPv4 zuständige docker-proxy-Prozess bei Zugriffen inaktiv bleibt. Wenn sie `strace` hingegen den für IPv6 zuständigen docker-proxy untersuchen lassen, sehen Sie dessen Systemaufrufe, mit denen er die Pakete weiterleitet.

Misstrauische Zeitgenossen können noch die Gegenprobe machen: In der Firewall-Chain „DOCKER“ legt Docker beim Start der Container für IPv4 und jeden Port eine DNAT-Regel an, die den Netzwerkverkehr für die Ports und an das gewählte Ziel weiterleitet (DNAT steht für Destination Network Address Translation). Löscht man diese Regeln, wird der für IPv4 zuständige docker-proxy-Prozess aktiv – `strace` gibt dann auch für IPv4 Systemaufrufe aus.

Wenn ein docker-proxy-Prozess eingehende Netzwerkanfragen auf die Ports der Container verteilt, hat das Nebenwirkungen: Bei den Containern nämlich kommt nicht die IP-Adresse des anfragenden Clients an, sondern nur eine Adresse des Hosts selbst (meist die für docker0 hinterlegte Adresse). In der Standardkonfiguration betrifft das nur IPv6-Verbindungen, weil der docker-proxy-Prozess für IPv4-Verbindungen gar nicht zum Zug kommt.

Es gibt aber durchaus auch Docker-Umgebungen, in denen diese Einschränkung auch für IPv4 gilt: Zuletzt begegnet ist uns das auf einem Synology-NAS-Gerät, während wir uns die Installation von Pi-hole auf solchen Geräten angesehen haben [1]. In dem Artikel haben wir das Verhalten fälschlicherweise der Standard-Bridge einer Docker-Installation zugeschrieben – es ist allerdings eine Eigenart der Synology-Implementierung, die durch fehlende Regeln in der Firewall stets docker-proxy-Prozessen die Arbeit überlässt.

Proxy-Tollpatsch

Kurzum: Für alle Dienste, die darauf angewiesen sind, dass im Container die IP-Adressen der realen

Clients ankommen, muss man einen Bogen um die Vermittlung durch die docker-proxy-Prozesse machen. Das betrifft zum Beispiel Mailserver, die per SPF (Sender Policy Framework) den Absender überprüfen, oder andere Anwendungen, die Clients unter anderem über die IP unterscheiden. Hilfreich sind die Adressen natürlich auch bei der Diagnose.

Die Docker-Entwickler haben vorgesehen, die docker-proxy-Prozesse zu entsorgen. Der Eintrag

```
{
"userland-proxy": false
}
```

in der Konfigurationsdatei /etc/docker/daemon.json genügt. Allerdings legt Docker dann nicht automatisch die erforderlichen IPv6-Firewall-Regeln an. Dazu müssen Sie einige Zeilen mehr ergänzen:

```
{
"userland-proxy": false,
"ipv6": true,
"fixed-cidr-v6": "fd00:111::/64",
"experimental": true,
"ip6tables": true
}
```

Mit der Option `ipv6` fordern Sie Docker auf, den Containern IPv6-Adressen zuzuteilen. Die stammen fürs standardmäßig aktive Bridge-Netzwerk aus dem mit `fixed-cidr-v6` genannten Adressbereich. Fehlt der, startet der Daemon nicht. Unser Beispiel nimmt einen zur lokalen Nutzung reservierten Block für Unique Local Addresses (ULAs); diese Adressen sind im Grunde das IPv6-Pendant der von Docker für IPv4 verwendeten privaten Adressen [2].

Mit `experimental` stellen Sie sicher, dass der Daemon die folgende Anweisung `ip6tables` überhaupt befolgt: Sie aktiviert erst das für IPv4 selbstverständliche Verhalten, dass für jeden per IPv6 erreichbaren, publizierten Port in einem Container eine Weiterleitung von der globalen IPv6-Adresse des Hosts auf die interne IPv6-Adresse des Containers eingerichtet wird.

Zudem erreichen Sie mit den genannten Optionen, dass auch aus dem Container heraus IPv6-Pakete ihren Weg in die weite Welt finden (so der Host selbst eine IPv6-Anbindung hat). Die Container mit ihren privaten Adressen verstecken sich dann per NAT hinter der IPv6-Adresse des Hosts. Für IPv6-Puristen ist das bäh, aber hier eine pragmatische, passende Lösung, die ziemlich genau das nachbildet, was Docker standardmäßig für IPv4 tut.

IPv6 experimentell

Die experimentelle IPv6-NAT-Implementierung weicht von der IPv4-Implementierung jedoch in einem Detail ab: Es werden keine Regeln für die DOCKER-USER-Chain erstellt. Das fällt normalerweise kaum auf. Es sei aber hier erwähnt, weil das zum Beispiel erforderlich ist, um mit fail2ban in die Firewall eingefügten Regeln Brute-Force-Attacken abzuwehren. Sollten Sie davon betroffen sein, können Sie sich an den iptables-Regeln für IPv4 orientieren und die DOCKER-USER-Chain für IPv6 entsprechend einrichten.

Übrigens: Lange bevor Docker selbst IPv6-Unterstützung mit NAT eingebaut hat, gab es diese schon von Robbert Klarenbeek in Form eines Docker-Containers (robbertkl/ipv6nat). Der mit umfangreichen Rechten zu startende Container wird so lange gepflegt, bis der experimentelle Support in Docker vollständig aufgeschlossen hat – heißt es im Ticket #65. Dort tragen die Nutzer zusammen, wo es in Docker noch hakt.

Würde man mit globalen IPv6-Adressen in den Containern arbeiten, ließe sich auf die experimentellen Features verzichten. Allerdings müsste man sich dann um ein geeignetes IPv6-Routing kümmern, bräuchte dafür mehrere IPv6-Netze und einige weitere IPv6-typische Dienste – unterm Strich nichts, was sich eben mal so einrichten ließe.

Und, was viel schwerer wiegt und die Nutzung von globalen IPv6-Adressen mit beliebigen Container-Images infrage stellt: Jeder in einem Container erreichbare Dienst respektive Port wäre unter der globalen IPv6-Adresse erreichbar, ohne dass er explizit exponiert werden muss. Das gilt immer, wenn vor den Containern nicht eine Firewall steht, die Zugriffe verhindert.

Es gibt noch einen weiteren guten Grund, den IPv6-Proxy von Docker durch Firewall-Regeln zu ersetzen: Der Server wird durch den Netzwerkverkehr weniger stark belastet. Denn während die Firewall vom Kernel bereitgestellt wird, läuft docker-proxy im Usermode – eingehende Anfragen verursachen in der Standardkonfiguration also einige performancezehrende Kontextwechsel zwischen Kernel- und Usermode mehr.

Traefik-Tricks

Wer jetzt denkt, mit einem Proxy wie Traefik fein raus zu sein und von all den Problemen nichts zu merken, täuscht sich: Steckt Traefik selbst in einem Bridge-

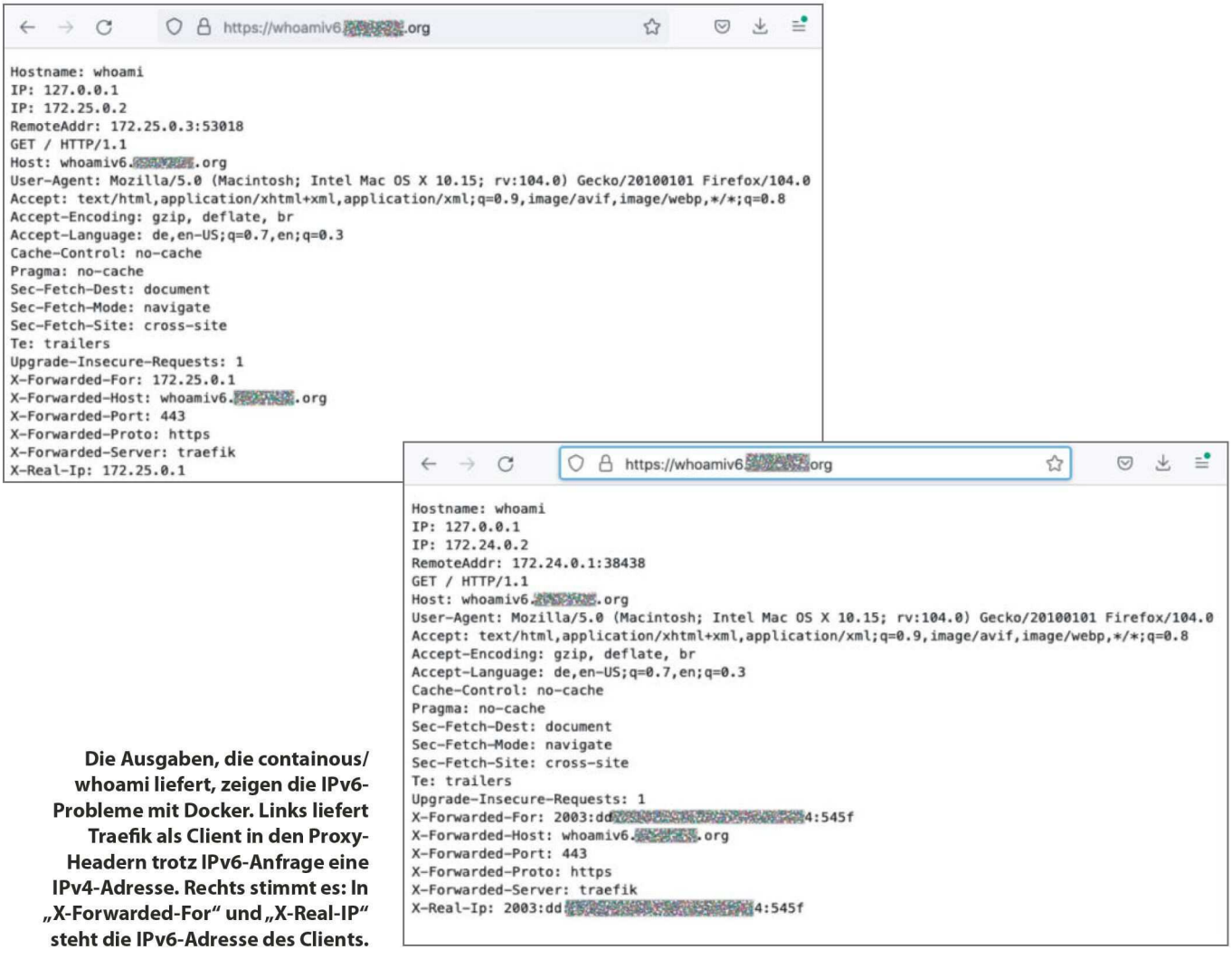
Literatur

[1] Peter Siering, **Speicher mit Filter**, Pi-hole oder AdGuard Home auf einem NAS einrichten, c't 20/2022, S. 76

[2] Dušan Živadinović, **Kleine Expedition**, IPv6-Grundlagen und Streifzug durch Ihr eigenes Netz, c't 7/2022, S. 56

Erwähnte Container und Tickets auf GitHub

ct.de/wp8n



Die Ausgaben, die containous/whoami liefert, zeigen die IPv6-Probleme mit Docker. Links liefert Traefik als Client in den Proxy-Headern trotz IPv6-Anfrage eine IPv4-Adresse. Rechts stimmt es: In „X-Forwarded-For“ und „X-Real-IP“ steht die IPv6-Adresse des Clients.

Netzwerk (das ist die übliche Konfiguration), landen in einer Standardinstallation von Docker auf Linux wie beschrieben nur die Adressen von IPv4-Zugriffen dort. IPv6-Zugriffe registriert Traefik mit der IPv4-Adresse des docker0-Interface. Das kann man im Antwort-Header mit containous/whoami sichtbar machen (siehe S. 24 und die Bilder oben).

Wenn Traefik allein die eingehenden Verbindungen verwaltet, ist es nicht notwendig, die experimentellen Features in Docker zu aktivieren. Es führt noch ein anderer Weg zu den realen IPv6-Adressen der Clients: Lassen Sie den Proxy im Host-Netzwerkmodus laufen. In dem Fall mischen sich weder docker-Proxy-Prozesse noch die Firewall ein. Traefik lauscht direkt auf den konfigurierten Ports und sieht so auch die realen Adressen anfragender IPv6-Clients und kann sie im Anfrage-Header korrekt übermitteln. So einfach kann eine Lösung auch sein. (ps) c't

Docker für Faule

Wer viel mit Docker-Containern auf der Kommandozeile arbeitet, hantiert schnell in unübersichtlich vielen Terminal-Fenstern. Lazydocker vereint Informationen über Container und wichtige Docker-Befehle unter einem Dach. Wir haben uns das mal angesehen.

Von **Niklas Dierking**

Wenn ein Docker-Container mal zickt, dann kann sich die Fehlersuche auf der Kommandozeile mühsam gestalten. Das gilt umso mehr, wenn man mehrere Container mittels Docker-Compose miteinander verdrahtet. Zunächst lässt man sich mit `docker ps` den Status der laufenden Container anzeigen: Der zickige Container startet laufend neu, da stimmt also etwas nicht. Als Nächstes wirft man mit `docker logs` einen Blick in den Container. Dabei springen mögliche Probleme ins Auge, also wird der Texteditor geöffnet und etwas in der `docker-compose.yml`-Datei angepasst. Jetzt schaut man dem Container mit `docker compose up` beim Starten über die Schulter, dafür ist jetzt aber das Terminal-Fenster belegt. Ärgerlich.

Das alles kostet Zeit und Nerven. Eine grafische Oberfläche hilft dabei, schnell die Situation im Container-Park zu überblicken. Mit Portainer oder Rancher gibt es dafür bereits mächtige Tools mit eigener Weboberfläche. Wer es eine Nummer kleiner mag, freut sich über Lazydocker. Die in Go geschriebene Anwendung fasst die wichtigsten Informationen über Ihre Container in einem Terminal-Fenster zusammen und sendet Befehle an den Docker-Daemon.

Sie navigieren Lazydocker mit der Maus oder Tastatur. Das Programm informiert über laufende Container und listet Images und Volumes auf. Für einzelne Container oder ganze Compose-Konfigurationen können Sie sich die Dockerfiles, Logs, Systemauslastung sowie laufende Prozesse anzeigen lassen. Besonders praktisch: Ein Kontextmenü, das sich mittels Druck auf die X-Taste öffnet, kann direkt mit dem Container interagieren und ihn etwa stoppen, neu starten, entfernen oder eine interaktive Shell öffnen. Auch ungenutzte Images lassen sich so leicht entfernen. Lazydocker ist größtenteils auf Deutsch übersetzt. Auf GitHub hat der Entwickler eine Übersicht über die Tastaturbefehle veröffentlicht.

Wer Lazydocker auf die eigenen Anforderungen zuschneiden möchte, bearbeitet die Konfigurationsdatei. Das funktioniert auch in Lazydocker selbst, indem man in der linken, oberen Ecke des Fensters auf „Projekt“ klickt und anschließend die E-Taste betätigt.

Lazydocker steht kostenlos im GitHub-Repository des Entwicklers zum Download bereit. Linux-Nutzer laden das aktuelle Release als Archiv herunter und verschieben die entpackte Binärdatei in das Verzeichnis /usr/local/bin. Danach ruft man den Docker-Helfer mit dem Befehl `lazydocker` auf. Versionen für Windows und macOS stehen ebenfalls zur Verfügung. Als Docker-spezifisches Werkzeug liegt Lazydocker auch selbst als Container-Image vor.

Insgesamt: Lazydocker greift Docker-Nutzern unter die Arme, die sich etwas Arbeit abnehmen lassen wollen, aber dafür keine ausgewachsene Container-Verwaltung mit Weboberfläche brauchen. (ndi) c't

Lazydocker

Docker-Helfer	
Hersteller, URL	Jesse Duffield, github.com/jesseduffield/lazydocker
Systemanf.	Docker (Windows, macOS, Linux)
Preis	kostenlos, Open Source (MIT Lizenz)

Bild: KI Midjourney

Von Docker Desktop auf Podman wechseln

Docker Desktop war über Jahre die unangefochtene Nummer Eins unter den Containerumgebungen auf Entwicklerrechnern. Doch Dockers neue Lizenzpolitik und Verbesserungen der Konkurrenz machen die Open-Source-Alternative Podman attraktiver. Der Wechsel geht verblüffend einfach.

Von **Jan Mahn**

Podman ist einst als Alternative zu Docker angetreten – mit kräftiger Unterstützung von Red Hat. Jahrelang kämpfte die Software um ihren Platz im Container-Mainstream, populär war sie bei zwei Zielgruppen: bei Linux-Nutzern aus dem Red-Hat- und Fedora-Umfeld und sicherheitsbewussten Anwendern, die besonderen Gefallen an „rootless containers“, also Containerbetrieb ohne Root-Rechte, gefunden hatten (siehe Artikel „Rootless-Container mit Podman betreiben“ auf S. 34). Alle anderen nutzten Docker Desktop.

Rosige Zeiten für Marktführer Docker Inc. eigentlich, doch nebenbei musste das Unternehmen Geld verdienen und verschärfte seine Lizenzbedingungen. Seit 31. Januar 2022 muss man zahlen, wenn man Docker Desktop in einem Unternehmen einsetzen

will, das mehr als 250 Mitarbeiter beschäftigt oder mehr als 10 Millionen US-Dollar Umsatz macht. Das gilt auch für staatliche Einrichtungen. Docker darf die Lizenzbedingungen ändern, obwohl Teile von Docker Desktop Open Source sind – aber eben nur Teile wie die Engine, die man auch weiterhin kostenlos im Unternehmen nutzen darf.

Bei der Alternative Podman sieht das Geschäftsmodell grundsätzlich anders aus: Mit Red Hat steht ebenfalls eine große Firma hinter der Entwicklung der Containerumgebung, die Software ist aber vollständig quelloffen und Red Hat will sein Geld mit Lizenzen und Support für Red Hat Enterprise Linux (RHEL) und der Kubernetes-Anwendungsplattform OpenShift verdienen, nicht mit Podman. Die Investition in Podman lohnt sich für die IBM-Tochter dennoch, schließlich kann das Containerökosystem, in dem auch OpenShift zu Hause ist, besser gedeihen, wenn es eine freie Containerentwicklungsplattform gibt. Denn dann hängt der Erfolg von Containern nicht von Docker Inc. ab.

Die Komponenten

Der Umstieg von Docker Desktop auf Podman ist nicht so schmerzhaft, wie man es von einem so radikalen Systemwechsel vielleicht erwarten würde. Podman macht unter der Haube zwar ein paar Dinge grundsätzlich anders, verhält sich nach außen aber in vielen Bereichen bewusst identisch. Die Befehle auf der Kommandozeile funktionieren wie unter Docker, statt `docker run` kann man `podman run` schreiben und Podman gibt Umsteigern in der Dokumentation direkt den Tipp, einfach in der Shell einen Alias von `docker` auf `podman` einzurichten. Wenn Sie nicht gleich vollständig wechseln wollen, können Sie beide Umgebungen auch parallel betreiben.

Im Alltag als Entwickler bemerkt man vom Umstieg wenig, nur das Einrichten des neuen Containerhosts ist aktuell noch mit etwas Mühe verbunden. Während Docker Desktop mit einem Installationsassistenten ausgeliefert wird, den man schnell durchklicken kann, muss man mehrere Podman-Komponenten (Podman selbst, ein GUI und einen Ersatz für Docker-Compose) installieren und teils aus verschiedenen Quellen zusammentragen, um denselben Funktionsumfang zu erreichen. Die grafische Oberfläche braucht man nicht unbedingt, sie liefert vor allem Einblicke, welche Container laufen und welche Abbilder geladen sind, mit Erweiterungen kann man zusätzliche Funktionen nachrüsten. Die nötigen Schritte zur Installation der Komponenten haben wir in den folgenden Abschnitten für Windows, Linux und macOS zusammengefasst. Für alle Betriebssysteme geht es anschließend im Abschnitt Ersteinrichtung weiter.

Umstieg unter Windows

Wenn von Containern die Rede ist, die mit Docker oder Podman gestartet werden, geht es fast immer um Linux-Container. Falls Sie zur eher kleinen Zielgruppe gehören, die Windows-Container mit Docker entwickelt und testet, ist Podman noch keine Alternative. Diese Funktion bekommen Sie aktuell exklusiv bei Docker.

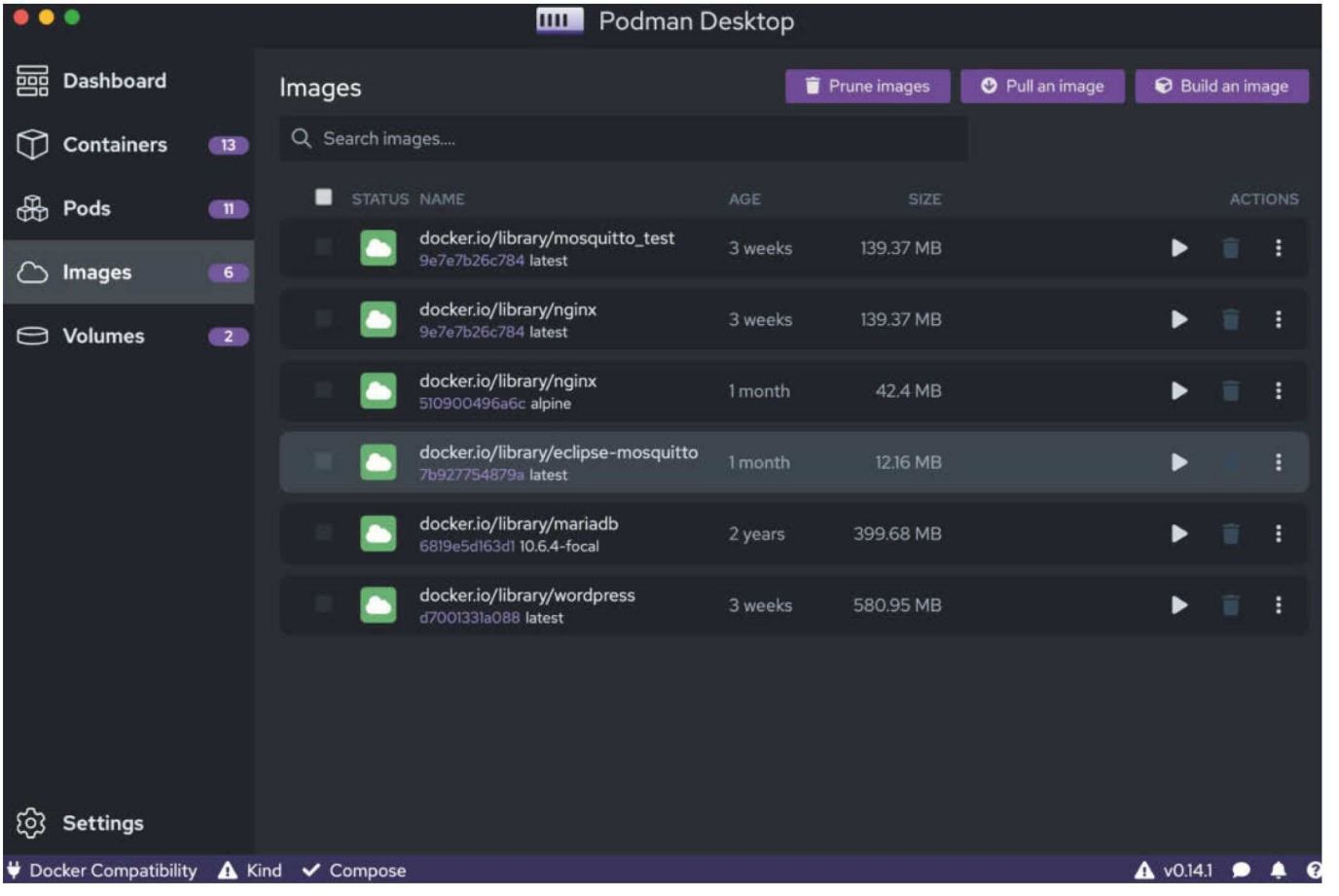
Wenn man einen Linux-Container startet, läuft darin eine Linux-Anwendung, aber kein Linux-Kernel – den bekommt der Container vom gastgebenden Betriebssystem. Auf einem Windows-PC, auf dem Linux-Container laufen sollen, fehlt ein solcher Kernel von Haus aus. Sowohl Docker Desktop als auch Podman bedienen sich an einer vergleichsweise jungen Windows-Funktion, dem Windows Subsystem for Linux in Version 2 (WSL2). Voraussetzung ist ein aktuelles Windows 10 oder 11 und die Virtualisierungsfunktionen des Prozessors müssen im BIOS aktiviert sein. Sollte das Windows selbst schon in einer virtuellen Maschine stecken, muss „Nested Virtualization“ hardwareseitig unterstützt und aktiviert sein.

Podman kann die Betriebssystemerweiterung WSL2 im Rahmen des Assistenten einrichten, schneller geht es aber, wenn Sie den Schritt vorab selbst erledigen. Was zu Beginn von Microsofts WSL-Experimenten noch fummelig war, gelingt jetzt mit einem einzigen Befehl auf einer Kommandozeile mit Administratorrechten:

```
wsl --install
```

Wenn Sie WSL ausschließlich für Podman nutzen und keine andere WSL-Distribution einrichten wollen (die unweigerlich Speicherplatz belegt), hängen Sie den Parameter `--no-distribution` an den Befehl an. Am Ende müssen Sie den Computer neustarten, anschließend ist er bereit für die Podman-Installation.

Den Installer finden Sie im GitHub-Repository des Projekts unter Releases (siehe ct.de/wu9u). Suchen Sie dort nach der aktuellen Version (als dieser Artikel entstand, war das 4.5.0), scrollen Sie bis zum Abschnitt Assets und laden Sie den Installer im .msi-Format oder als .exe-Datei herunter. Beide Varianten führen zum selben Ergebnis; über das MSI-Paket freuen sich besonders Administratoren, die



Podman Desktop verrät unter anderem, welche Abbilder bereits auf der Festplatte liegen.

Podman auf vielen Maschinen automatisch bereitstellen müssen.

Nach Podman können Sie Podman Desktop installieren. Der Installer liegt unter der Adresse podman-desktop.io bereit. Wenn Sie die Seite mit einem Windows-Rechner besuchen, sollten Sie die Windows-Ausgabe direkt zum Download angeboten bekommen. Der Installer verrichtet schnell und ohne viele Nachfragen seinen Dienst. Rechts im Infobereich der Taskleiste finden Sie anschließend einen neuen Eintrag für Podman, über den Sie die Oberfläche öffnen.

Um loscontainern zu können, brauchen Sie anschließend eine WSL-Umgebung für Podman. Wer auf Podman Desktop verzichtet, wo ein Klick auf den Initialisieren-Button genügt, muss das mit zwei Kommandozeilenbefehlen erledigen:

```
podman machine init
podman machine start
```

Der erste Befehl richtet eine WSL-Instanz mit einem minimalen Fedora-Linux ein, der zweite startet sie. Wenn Sie bisher Docker genutzt haben und vollständig umsteigen, Docker also nicht mehr parallel betreiben wollen, sollten Sie den Docker-Daemon beenden, bevor Sie den zweiten Befehl absetzen. Dann kann Podman automatisch den Unix-Socket

beanspruchen, den bisher Docker abgehört hat. Warum das nützlich ist, lesen Sie im Abschnitt Ersteinrichtung. Sollten Sie die Reihenfolge nicht eingehalten haben, ist das aber kein Problem. Schalten Sie den Docker Daemon ab und setzen `podman machine start` erneut ab.

Eine Windows-Besonderheit ist traditionell der Umgang mit Pfadangaben. Sie funktionieren anders als in unixoiden Betriebssystemen. Das Einrichten von Volumes, bei denen ein Windows-Pfad in einen Container übergeben wird, war in den Anfangszeiten von Docker eine Qual. Mit Podman und dem aktuellen WSL hat die Aufgabe ihren Schrecken verloren. Folgende Beispiele funktionieren problemlos, beide bringen ein Web-Verzeichnis aus dem Nutzerverzeichnis des fiktiven Nutzers me auf Laufwerk c:\ in einen Nginx-Container:

```
podman run -v c:\Users\me\web:↵
↳/usr/share/nginx/html nginx
podman run -v /c/Users/me/web:↵
↳/usr/share/nginx/html nginx
```

Umstieg unter macOS

Für macOS-Nutzer (mit Intel- oder Apple-Prozessor) ist der Paketmanager Homebrew (zu finden über brew.sh) der komfortabelste Weg, der auf vielen Admin- und Entwicklermaschinen ohnehin zum Standard gehört. Damit ist die Installation in wenigen Minuten und mit einem Befehl erledigt:

```
brew install podman
```

Wenn Sie auf eine grafische Oberfläche nicht verzichten wollen, installieren Sie Podman Desktop direkt hinterher:

```
brew install podman-desktop
```

Das Podman-GUI finden Sie nach der Installation im Programme-Ordner, außerdem nistet es sich in macOS in der Menüleiste oben rechts ein.

Im Anschluss brauchen Sie noch zwei Befehle, die im Hintergrund eine virtuelle Maschine einrichten und diese starten. Diese abgespeckte Linux-VM ist notwendig, damit Linux-Container auf einem Mac mit macOS laufen können. Zwar ist macOS ein unixoides Betriebssystem, aber es hat trotzdem keinen Linux-Kernel, den die Prozesse in Containern zum Laufen brauchen. Die VM schließt die Lücke, bleibt im Alltag aber weitestgehend unsichtbar, wie auch bei Docker Desktop, das unter macOS nach demselben Prinzip arbeitet. Die VM mit der grafischen Oberfläche zu initialisieren hat in unserem Testlauf nicht funktioniert, also war der Umweg über die Kommandozeile fällig:

```
podman machine init
podman machine start
```

Um den Zustand dieser Maschine zu überwachen und sie zu stoppen (um zum Beispiel Ressourcen zu sparen, wenn man nicht containert), gibt es die Befehle `podman machine ls` und `podman machine stop`.

Zum Abschluss der Einrichtung gibt es noch einen kleinen und lohnenswerten Schritt: Podman kann sich wie der Docker-Daemon verhalten, das API nachahmen und auch denselben Unix-Socket abhören – dann müssen Sie sich aber gegen einen Parallelbetrieb entscheiden. Software, die bisher mit Docker gesprochen hat, kommuniziert dann mit Podman. Um die Funktion zu aktivieren, stoppen Sie zuerst den Docker-Daemon. Im Podman-GUI klicken Sie dann unten links auf „Docker Compatibility“. Nach einem Neustart von Podman hat es die Rolle von Docker übernommen; wie Sie diese Funktion nutzen, folgt im Abschnitt Ersteinrichtung. Wenn Sie auf die grafische Oberfläche verzichten, aktivieren Sie die Docker-Kompatibilität mit:

```
sudo podman-mac-helper install
```

Anschließend braucht die Maschine einen Neustart (mit `podman machine stop` und …`start`).

Umstieg unter Linux

Mit den meisten Linux-Distributionen ist ein Umstieg auf Podman sehr schnell erledigt, in vielen offiziellen Paketquellen ist Podman enthalten. Unter Arch Linux, Alpine, Ubuntu, Debian (gilt auch für Raspberry Pi OS), openSUSE und Fedora heißt das Paket schlicht `podman`. Greifen Sie zum Paketmanager Ihrer Distribution und starten Sie die Installation. Unter Ubuntu und Debian zum Beispiel mit:

```
sudo apt update
sudo apt install podman
```

In der offiziellen Dokumentation finden Sie Installationshinweise für exotischere Distributionen, für ältere Podman-Versionen und Rezepte für Selbstkompilierer (siehe ct.de/wu9u).

Die grafische Oberfläche von Podman Desktop ist auf einem Linux-Desktop verzichtbar: Anders als unter Windows und macOS gibt es unter Linux keine Virtualisierungsschicht, die man darüber verwalten könnte. Wer ihn nutzen will, bekommt den Containermanager derzeit am bequemsten per Flatpak ins System:

```
flatpak install --user flathub ↵
↳io.podman_desktop.PodmanDesktop
```

Wer Flatpak meidet, muss sich die ausführbare Datei als Tar-Archiv aus dem GitHub-Repository herunterladen, entpacken und ausführen. Die offizielle Dokumentation beschreibt das im Detail (siehe ct.de/wu9u).

Ersteinrichtung

Einmal installiert, ist es an der Zeit, sich an die Podman-Besonderheiten zu gewöhnen. Podman arbeitet von Haus aus ohne einen Daemon, der rund um die Uhr mit Root-Rechten läuft, Befehle über ein API entgegennimmt und Container startet. Stattdessen erzeugt ein Podman-Aufruf einen Prozess, der den Container als Kind-Prozess startet; fork-exec heißt das Konzept in der Linux-Welt. Und unter Linux funktioniert diese reine Lehre auch. Im nachfolgenden Artikel „Rootless-Container mit Podman betreiben“ ab Seite 34 erfahren Sie im Detail, wie Sie dieses Konzept ausnutzen, um Container sicherer und ohne Root-Rechte zu starten und welche Tücken es mit rootfreien Containern unter Linux gibt.

Unter macOS und Windows sieht die Welt ein bisschen anders aus. Auf beiden Systemen gibt es keine reine fork-exec-Lehre, weil noch eine Abstraktionsschicht (VM oder WSL) im Spiel ist. Diese Abstraktionsschicht arbeitet zunächst ohne Root-Rechte, was sich spätestens dann bemerkbar macht, wenn Sie einem Container einen TCP/UDP-Port kleiner als 1024 zuweisen wollen. Ändern kann man das Verhalten, indem man die Abstraktionsschicht stoppt und mit Root-Rechten startet:

```
podman machine stop
podman machine set --rootful
```

Im Gegenzug für den Mehraufwand mit der Abstraktionsschicht bekommt man auf diesen Systemen ein API, das das Docker-API nachahmt. Das führt zu einer interessanten Situation: Solange das Docker-CLI installiert ist, kann man mit diesem fast nahtlos weiterarbeiten, obwohl es mit Podman spricht. Das zeigt sich, wenn man den Befehl `docker version` absetzt. Er gibt zwei Abschnitte zurück: Im ersten Teil stehen Details zum CLI selbst, im zweiten Teil taucht Podman als Server auf. Auf diesem Weg kann man auch Docker-Compose-Dateien ausführen, denn der Unterbefehl `docker compose up` ruft jetzt das von Podman nachgebaute Docker-API auf und arbeitet weiterhin, nun aber mit Podman statt Docker.

Für alle drei Betriebssysteme gibt es eine Compose-Alternative, die ganz ohne Software aus dem Hause Docker auskommt: `podman-compose`. Das ist eine in Python geschriebene Kommandozeilenanwendung, die Compose-Dateien liest und direkt in Podman-Befehle übersetzt – also ganz ohne das nachgebildete Docker-API auskommt. Um die Software zu installieren, muss Python 3 auf der Maschine installiert sein, zusammen mit dem Python-Paketmanager pip. Dann reicht auf allen Betriebssystemen ein Befehl für die Installation:

```
pip install podman-compose
```

Fedora-Nutzer können auch zu ihrem Paketmanager greifen:

```
sudo dnf install podman-compose
```

Podman-Compose ahmt die Funktionsweise von Docker-Compose weitestgehend nach und ist für den Ein- und Umstieg gut geeignet, weil man bestehende Zusammenstellungen nicht neu schreiben muss. Mit `podman-compose up` und `down` müssen Sie sich kaum umgewöhnen.

Neue Welten

Während der ersten Wochen mit Podman werden Sie immer wieder auf kleine Problemchen mit Berechtigungen, Netzwerken und Volumes stoßen, weil Podman eben nicht Docker ist und auch nicht sein will – wenn Sie diese gelöst haben, gibt es im Podman-Universum Funktionen zu entdecken, die Docker nicht zu bieten hat. Da ist zum Beispiel die Möglichkeit, Container wie in Kubernetes zu Pods zusammenzufassen und Zusammenstellungen statt in Compose-YAML in Kubernetes-Syntax als Pod und Deployment zu definieren (siehe Artikel „Der Lernpfad zum Kubernetes-Kenner“ auf S. 38). Die Marschrichtung von Podman ist klar: Die Software will eine Entwicklungs- und Testumgebung bereitstellen, um Container für den Betrieb in Kubernetes vorzubereiten – ganz ohne Abhängigkeit von Docker Inc. (jam) c't

Podman-Dokumentation
ct.de/wu9u

Know-How statt Hype

Mit KI-Tools effektiv arbeiten

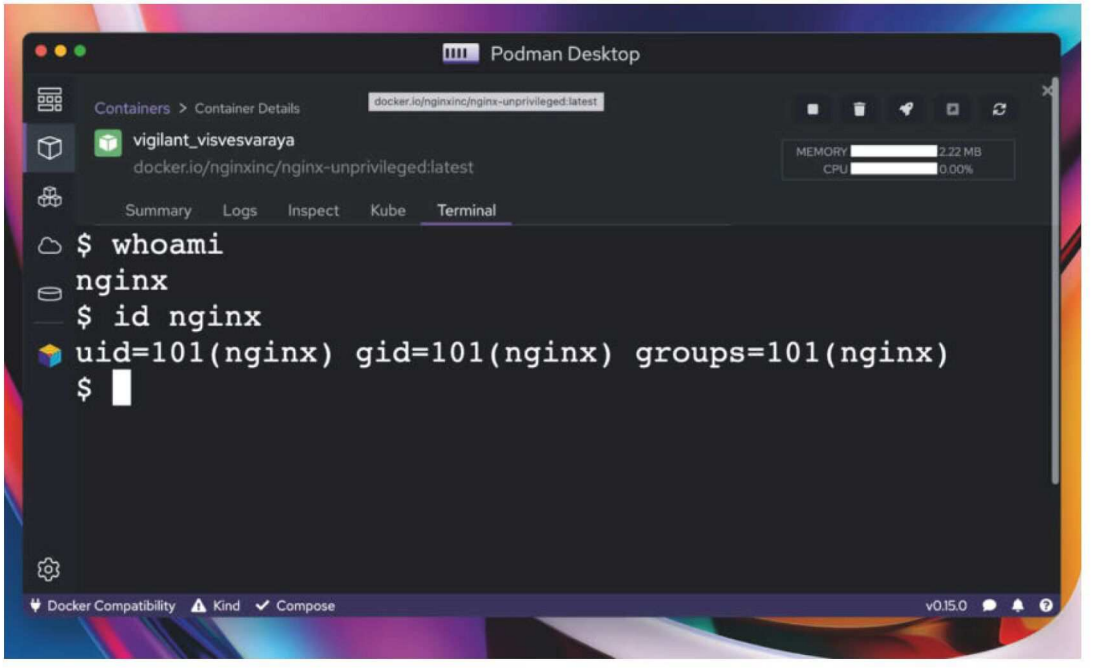
Heft + PDF mit 29 % Rabatt

Die Nachrichten über revolutionäre KI-Lösungen überschlagen sich täglich. Wie soll man da den Überblick behalten? Mit Tests und Praxistipps erklären wir im c't-Sonderheft, was heute schon geht sowie Ihnen bei der Arbeit hilft und wo Sie den Maschinen noch Zeit zum Reifen geben sollten.

- ChatGPT zwischen wirtschaftlicher Effizienz und menschlichem Wunschdenken
- Bilder-KI Stable Diffusion lokal installieren und betreiben
- Textgeneratoren für jeden Zweck
- Sprachmodelle mit Suchmaschinen koppeln
- Vier KI-Komponisten im Test
- ChatGPT als Hacking-Tool

Heft für 14,90 € • PDF für 12,99 € • Bundle Heft + PDF 19,90 €

shop.heise.de/ct-chatgpt



Rootless-Container mit Podman betreiben

Wenn Admins von Containern sprechen, dann meinen sie meistens Docker. Podman ist eine alternative Container-Engine, die mit den gleichen Befehlen arbeitet und auf Daemon und Root verzichtet. Wir zeigen, wie Sie mit Podman „Rootless"-Container betreiben, ohne sich umzugewöhnen.

Von **Niklas Dierking**

Dank Container-Engines wie Docker und prall gefüllten Container-Registries mit fertigen Images für jeden erdenklichen Zweck dauert es heutzutage wenige Sekunden, eine Webanwendung oder einen Webserver aufzusetzen. Was soll schon schiefgehen? Immerhin sehen die Prozesse im Container ja nur dessen Dateisystem und sind dank Kernel-Namespaces vom Host isoliert. Wenn Sie jedoch mit dem Befehl `docker run` ein Container-Image aus der Docker-Registry ziehen und den Container starten, dann läuft dieser gewöhnlich mit Root-Rechten. Das heißt, der Benutzer root im Container entspricht root auf dem Host.

Es könnte einem geschickten Angreifer gelingen, aus einem Container auszubrechen, indem er eine Schwachstelle im Container ausnutzt. Dann ist der

Host in Windeseile übernommen. Weil der Docker-Daemon, der alle Fäden in der Hand hält, mit Root-Rechten läuft, kann man damit auch das Wurzelverzeichnis / in einen Container hineinreichen und so Dateiberechtigungen aushebeln. Alle Nutzer, die Zugriff auf den Docker-Daemon haben, verfügen also de facto über Systemverwalterrechte. Ein Albtraum für Administratoren.

Rootless-Container, also Container, die nicht als root laufen, verkleinern die Angriffsfläche erheblich. Praktisch: Solche Container können von gewöhnlichen Nutzern gestartet werden. Das ist wichtig für Systeme, die sich mehrere Nutzer für die Containerei teilen, etwa in der Forschung oder auf einer geteilten Entwicklungsmaschine. Der Docker-Konkurrent Podman aus der Red-Hat-Welt läuft standardmäßig im Rootless-Modus. Podman gaukelt Prozessen mit Root-Rechten im Container vor, dass Sie als root laufen. In Wahrheit segeln sie aber auf dem Host unter der Flagge eines gewöhnlichen Benutzers ohne besondere Privilegien, außer Sie starten Podman absichtlich als root. Dazu später mehr.

Installation und Vorbereitung

Wenn Sie mit Docker vertraut sind, dann werden Sie sich bei Podman direkt zu Hause fühlen. Wie Docker folgt die Software den Regeln der OCI (Open Container Initiative) für das Bauen und Ausführen von Images. Die Podman-Befehle sind nahezu identisch mit denen von Docker. Wenn Sie wollen, können Sie sogar einfach ein Alias setzen, das von `docker` auf `podman` verweist. Von Docker unterscheidet sich Podman durch den Verzicht auf einen Daemon. Die Software nutzt stattdessen das fork-exec-Modell, um Prozesse zu starten und arbeitet eng mit systemd zusammen. Ähnlich wie beim Container-Orchestrierer Kubernetes lassen sich mehrere Container zu „Pods“ zusammenfassen, die sich dann Ressourcen teilen. Mehr zur grundlegenden Funktionsweise von Podman lesen Sie in [1]. Wie Sie von Docker Desktop auf Podman umsteigen, haben wir im Artikel ab Seite 28 aufgeschrieben. Der folgende Absatz erklärt die Installation auf einem frischen Linux-Host.

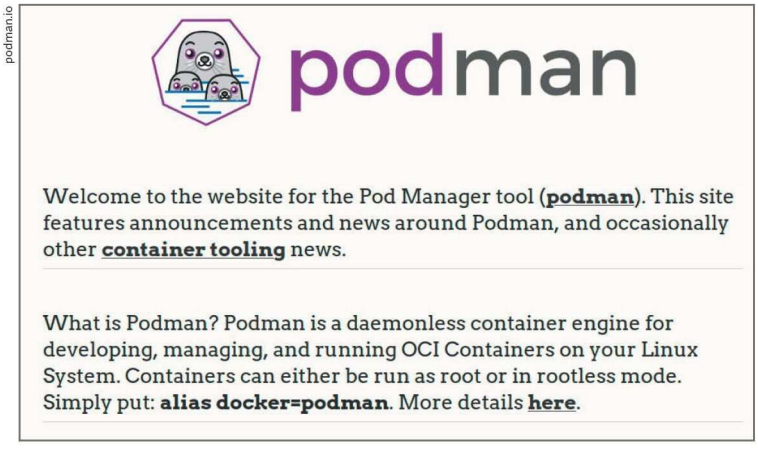
Installieren Sie Podman und die Pakete slirp4netns und fuseoverlayfs mit dem Paketmanager Ihrer Distribution. Auf unserem Testsystem kommt Fedora 38 mit dnf zum Einsatz, Sie können zum Ausprobieren aber auch Podman Desktop benutzen:

```
sudo dnf install podman slirp4netns \
fuseoverlayfs
```

Podman ist auch für Windows und macOS verfügbar. Podman nutzt dann das Windows Subsystem for Linux oder spannt – ähnlich wie Docker Desktop – eine schmale virtuelle Maschine ein, die die Container-Runtime beherbergt.

Jetzt müssen Sie sicherstellen, dass Ihr System die nötigen Anforderungen für Rootless-Container erfüllt. Podman bedient sich für Rootless-Container einer Funktion des Linux-Kernels namens „user namespaces“. User namespaces ermöglichen, dass

podman.io



Der Docker-Konkurrent Podman kommt ohne Daemon aus und macht „Rootless“-Container zum Standard.

Prozesse innerhalb eines Namespace mit anderen UIDs und GIDs (User Identifier, Group Identifier) laufen als außerhalb. Damit gaukelt Podman Prozessen im Container vor, sie verfügen über Root-Rechte. Dafür müssen dem unprivilegierten Podman-Benutzer eine Reihe von UIDs und GIDs in den Dateien /etc/subuid und /etc/subgid zugeteilt sein. Die Dateien legt das Paket shadow-utils an, das in Fedora standardmäßig enthalten ist.

In /etc/subuid und /etc/subgid müssen pro Zeile ein Nutzername aus /etc/passwd, eine Start-UID und der verfügbare UID-Bereich in folgendem Format eingetragen sein:

```
BENUTZER:START-UID:UID-BEREICH
```

Wenn Sie in Fedora 38 neue Benutzer mit dem Befehl `useradd` anlegen, vergibt das Programm automatisch einen Bereich von 65536 UIDs an jeden Benutzer, wie sich mit `cat /etc/subuid` prüfen lässt:

```
ndi:100000:65536
cttest:165536:65536
cttest2:231072:65536
cttest3:296608:65536
```

Bei bestehenden Benutzern müssen Sie die Werte unter Umständen manuell anpassen. Das erledigen Sie entweder mit einem Texteditor oder mit folgendem Befehl:

```
usermod --add-subuids 100000-165535 ↵
↳--add-subgids 100000-165535 cttest
```

Achten Sie darauf, dass die UID-Bereiche sich nicht mit bestehenden UIDs überlappen. Der Benutzer bekäme sonst Zugriff auf Dateien, die der fremden UID zugeordnet sind.

Wer bin ich wo?

Nachdem Sie diese Vorarbeiten geleistet haben, können Sie mit Podman Container auf verschiedene Weisen mit und ohne Root-Rechte betreiben. Bei der Konstellation, die Dockers Standardkonfiguration am nächsten kommt – und von der Sie sich verabschieden wollen –, führen Sie Podman als root aus. Die Prozesse im Container laufen ebenfalls als root, also entspricht root im Container auch root auf dem Host. Das lässt sich gut veranschaulichen, indem Sie einen Beispielprozess wie `sleep` in einem Container anschieben:

```
[root@fedora] podman run -d --rm ↵
↳registry.access.redhat.com/↵
↳ubi9-micro sleep 1000
```

Dann prüfen Sie mit dem Befehl `ps -ef n | grep "sleep 1000"`, wem der Prozess auf dem Host zugeordnet ist. Wie erwartet läuft der Prozess mit der UID 0 (root), also innerhalb und außerhalb des Containers als root.

Das geht auch sicherer: Für Podman im Rootless-Modus rufen Sie Podman schlicht mit einem unprivilegierten Benutzer auf:

```
[cttest@fedora] podman run -d --rm ↵
↳registry.access.redhat.com/ubi9-↵
↳micro sleep 1000
```

Der Prozess läuft zwar im Container als root, ist auf dem Host jedoch der UID 1001 des unprivilegierten Nutzers cttest zugeordnet, wie die Ausgabe von `ps -ef n | grep "sleep 1000"` verrät:

```
1001    3062    3057  0 02:34 ? ↵
↳ Ss 0:00 /usr/bin/coreutils ↵
↳--coreutils-prog-shebang=sleep ↵
↳/usr/bin/sleep 10000
```

Durch den Rootless-Betrieb des Containers haben Sie die Angriffsfläche bereits um ein gutes Stück verkleinert. Wenn Sie noch mehr auf Nummer sicher gehen wollen, dann starten Sie den Container erneut als unprivilegierter Benutzer und ergänzen beim Aufruf von `podman run` die Option `-u` mit einer UID eines normalen Benutzers, beispielsweise 1001 für cttest:

```
[cttest@fedora] podman run -d -u 1001↵
↳ --rm registry.access.redhat.com/↵
↳ubi9-micro sleep 1000
```

Bei dieser Methode („rootless als non-root-user“) laufen Prozesse im Container ebenfalls als unprivilegierter Benutzer. Das bietet potenziellen Angreifern die kleinste Angriffsfläche und schränkt die Rechte der Prozesse im Container größtmöglich ein. Der Befehl `ps -ef n | grep "sleep 1000"` zeigt, dass der Prozess auf dem Host einer UID aus dem Bereich zugeordnet ist, den Sie in der Konfiguration in /etc/subuid festgelegt haben.

Es gibt aber einen Haken: Viele Container-Images sind darauf angewiesen, dass die Prozesse innerhalb des Containers als root laufen. Images, die mit eingeschränkten Rechten klarkommen, erken-

```
[root@ndilin2 ~]# id
uid=0(root) gid=0(root) Gruppen=0(root) Kontext=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[root@ndilin2 ~]# podman run -d --rm registry.access.redhat.com/ubi9-micro sleep 1000
f6c3f796ee37b9832af952de4a16d27302eb9cda2599cc124560c7274ba90b14
[root@ndilin2 ~]# ps -ef n | grep "sleep 1000"
      0   40340   40337  0 16:34 ?        Ss     0:00 /usr/bin/coreutils --coreutils-prog-shebang=sleep /usr/bin/sleep 1000
```

Root im Container entspricht root auf dem Host: Dieses Sicherheitsrisiko lässt sich leicht vermeiden, indem Sie Podman als unprivilegierter Benutzer starten.

```
[cttest@fedora ~]$ id
uid=1001(cttest) gid=1001(cttest) Gruppen=1001(cttest) Kontext=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[cttest@fedora ~]$ podman run -d -u 1001 registry.access.redhat.com/ubi9-micro sleep 1000
ee3fb4d1cc1c62672dfb54f6c1e2ac2938e9f0fdecb8f163a825855bd4616187
[cttest@fedora ~]$ podman exec -it ee3fb4d1cc1c62672dfb54f6c1e2ac2938e9f0fdecb8f163a825855bd4616187 /bin/bash
bash-5.1$ id
uid=1001(1001) gid=0(root) groups=0(root),1001(1001)
bash-5.1$ exit
exit
[cttest@fedora ~]$ ps -ef n | grep "sleep 1000"
  166536   59573   59570  0 19:43 ?        Ss     0:00 /usr/bin/coreutils --coreutils-prog-shebang=sleep /usr/bin/sleep 1000
```

Der Ansatz „rootless als non-root-user" bietet die größte Sicherheit, funktioniert aber nicht mit sämtlichen Container-Images.

nen Sie oft daran, dass Sie den Zusatz „unprivileged" im Namen tragen, beispielsweise der Webserver nginxinc/nginx-unprivileged (siehe ct.de/wd4s).

Stolperfallen

Der Rootless-Modus von Podman bietet zwar mehr Sicherheit, erfordert aber auch einige Einschränkungen. Wenn Sie Podman als unprivilegierter Benutzer ausführen, können die Container keine TCP-Ports nutzen, die kleiner als 1024 sind. Das betrifft beispielsweise die für Webserver wichtigen Ports 80 und 443. Als Workaround weisen Sie dem Container einen unprivilegierten Port wie 8080 zu und spannen dann einen Reverse-Proxy wie Traefik vor den Webserver, der Anfragen an den korrekten Port durchleitet. Alternativ können Sie auch die Spanne unprivilegierter Ports anpassen:

```
sysctl net.ipv4.ip_unprivileged_port_start=443
```

Der Befehl verschafft Rootless-Containern Zugriff auf alle Ports ab 443.

Beim Umstieg auf Rootless-Container lauern noch einige weitere Fallstricke, von denen Sie aber viele mit wenigen Befehlen beseitigen können, beispielsweise um aus Rootless-Containern andere Hosts anzupingen. Einschränkungen listen die Podman-Entwickler in einem GitHub-Repository auf, das wir unter ct.de/wd4s verlinkt haben. Weil die Entwickler den Rootless-Modus von Podman als ein zentrales Feature ansehen, dürfte sich die Situation in kommenden Versionen weiter verbessern.

Fazit

Sie haben jetzt das nötige Grundwissen, um unprivilegierte Container mit Podman zu betreiben und kennen die wichtigsten Fallstricke. Schwachstellen, die es Prozessen erlauben, aus Containern beziehungsweise Namespaces auszubrechen, sind zwar selten, kommen aber immer wieder vor (siehe ct.de/wd4s). Wenn Sie Ihre Container mit Podman im Rootless-Modus ausführen, machen Sie es einem potenziellen Angreifer deutlich schwerer, den Host zu übernehmen. (ndi) c't

Literatur

[1] Thorsten Leemhuis, **Linux-Container 2.0.**, Podman & Co. – mehr als ein transparenter Ersatz für Docker, c't 21/2019, S. 130

Beispiel-Images, Container-Schwachstellen

ct.de/wd4s

Der Lernpfad zum Kubernetes-Kenner

Kubernetes-Experten sind gefragt und viele Docker-Nutzer würden die andere Seite des Container-Universums gern mal kennenlernen – wäre das Ökosystem nicht so groß und undurchsichtig. Mit unserer ausführlichen Praxis-Reihe gelingt der Einstieg: Der erste Teil zeigt, wie Sie aus drei Linux-Servern einen Cluster bauen.

Von **Jan Mahn**

Bild: Albert Hulm

Das Softwareprojekt ist zu groß geworden für einen einzigen Docker-Server, Ihre Chefs erwarten von Ihnen jetzt Kubernetes-Erfahrung oder Sie wollen aus eigenem Antrieb verstehen, wie man seine Container mit der Software betreibt, die auch Schwergewichte wie Netflix, Spotify und Banken im Einsatz haben. Gründe, sich heute an den Einstieg in Kubernetes zu wagen, gibt es viele – Voraussetzung ist lediglich ein souveräner Umgang mit Docker oder einer anderen Container-Umgebung wie Podman. Wenn Sie noch nicht überzeugt sind, warum Sie Kubernetes brauchen und lernen sollten, finden Sie Argumente im Kasten „Darum Kubernetes“ auf Seite 40. Aber ohne, dass man einige Monate lang Container betrieben, Abbilder heruntergeladen und eigene gebaut hat, sollte man die Finger von Kubernetes lassen; Frust wäre garantiert. Eine Einführung in Docker und den aktuellen Stand lesen Sie im Artikel „Container verstehen und loslegen“ auf Seite 8.

Auch für erfahrene Docker-Nutzer führt der naheliegendste Weg in die Kubernetes-Welt leider schnell in eine Sackgasse. Beim ersten Blick auf die offizielle Kubernetes-Dokumentation wird man ziemlich zuverlässig erschlagen. Die liegt unter docs.kubernetes.io und wird später ein zuverlässiger Begleiter. Wie Sie vielleicht schon mitbekommen haben, stammt Kubernetes ursprünglich aus dem Hause Google und wird jetzt als branchenübergreifendes Open-Source-Projekt entwickelt. Daher arbeitet ein Team aus Dokumentationsprofis daran, die Texte auf dem aktuellen Stand zu halten und leistet gute Arbeit. Die Doku verrät jedes Detail und ist ein unverzichtbares Nachschlagewerk, denn auswendig lernen kann niemand alle Funktionen von Kubernetes. Für Einsteiger ist dieses Werk jedoch keine Empfehlung – das liegt auch daran, dass Sie neben Kubernetes auch gleich ein ganzes Ökosystem aus Open-Source-Projekten kennenlernen müssen, die im Zusammenspiel mit Kubernetes funktionieren. Und oft gibt es auch mehrere Projekte, die dasselbe Problem lösen. Die Kubernetes-Doku allein enthält also nur einen Teil der Wahrheit. Im Ökosystem gibt es aber so viele Pfade und Verzweigungen, dass man sich allzu leicht verlaufen kann.

Eigene Erfahrungen statt Theorie

Diese Artikelreihe möchte einen möglichen Weg durch das Profi-Container-Dickicht aufzeigen. Nicht den einzigen Weg und sicher nicht den besten Weg für alle erdenklichen Umgebungen, aber einen, der sich für Docker-Kenner bewährt hat. Wo es angebracht ist, erhalten Sie Hinweise auf alternative Routen. Im Mittelpunkt steht das Ausprobieren und Nachbauen: Anhand einer Anwendung, die aus einer Ein-Server-Docker-Umgebung in die Kubernetes-Welt umziehen soll, lernen Sie Kubernetes-Konzepte, Werkzeuge aus dem Ökosystem und erprobte Lösungsansätze kennen. Auf dem Weg verinnerlichen Sie Begriffe und Kommandozeilenbefehle ganz automatisch.

Kubernetes allein ist nicht der Schlüssel zum Erfolg – es ist das Ökosystem aus Open-Source-Projekten. Die Landkarte der Cloud Native Computing Foundation (landscape.cncf.io) zeigt, was es im Kubernetes-Universum alles zu entdecken gibt.

Darum Kubernetes

Kubernetes ist weit mehr als eine Docker-Alternative, die im Cluster-Betrieb läuft. Selbst im Ein-Server-Betrieb kann Kubernetes weit mehr als Docker: Zunächst sind da die Konfigurationsmöglichkeiten, die den Lebenszyklus eines Containers von Anfang bis Ende kontrollierbar machen. Was in einer Docker-Compose-Datei eine Zeile ist, kann man bei Bedarf in einer Kubernetes-YAML-Datei oft auch in 20 Zeilen haarklein steuern. Sie haben zum Beispiel Ärger mit der Startreihenfolge Ihrer Container, die voneinander abhängen, und die Werkzeuge von Docker reichen nicht aus, dass sie korrekt aufeinander warten? Kubernetes hat Mittel dagegen. Durch diese Steuerung des Containerlebenszyklus sind auch perfekte Rolling Updates kein Problem mehr. Einmal richtig eingestellt, können Sie Anwendungen aktualisieren, ohne dass Nutzer einen Ausfall bemerken. Mit etablierten Werkzeugen wie Helm wird auch das Installieren und Weitergeben von Containerzusammenstellungen viel einfacher als mit Docker-Compose-Dateien.

Im Hintergrund stellt Kubernetes eine Programmierschnittstelle bereit, auf die Fernsteuerungssoftware von außen und auch Container selbst zugreifen können. Neben den Zuständen von Containern kann das API auch alle Formen von Konfigurationen verwalten – und anders als der Docker-Socket hat es eine Benutzer- und Berechtigungsverwaltung.

Die Voraussetzungen zum Nachvollziehen dieser Einführung sind für Administratoren und Entwickler mit etwas Linux-Erfahrung keine unüberwindbare Hürde: Sie brauchen drei (virtualisierte) Server, die bestenfalls über öffentliche IP-Adressen im Internet ansprechbar sind und sich – sofern möglich – auch über ein internes Netz erreichen. Außerdem eine Domain, für die Sie Subdomains verwalten können. Nur dann können Sie später auch Experimente mit TLS und der Zertifikatsbeschaffung nachvollziehen.

Theoretisch könnten Sie auch mit einer einzigen Maschine Ihre Kubernetes-Karriere beginnen und selbst die lokale Entwicklermaschine mit installiertem Docker Desktop reicht aus, um einen Kubernetes-Cluster zu simulieren. Wenn Sie unter Windows, macOS und neuerdings auch Linux den Einstellungsdialog von Docker Desktop öffnen, können Sie beim Menüpunkt Kubernetes einen Single-Node-Cluster hochfahren. Die Funktion richtet sich vor allem an Entwickler, die testen müssen, wie sich ihr Container unter Kubernetes-Bedingungen verhält. Zum Lernen der Grundlagen ist das aber nicht die beste Wahl, weil Kubernetes erst im richtigen Cluster spannend wird.

In diesem ersten Artikel soll es nicht um Anwendungsentwicklung in Containern gehen, sondern um den Bau eines richtigen Clusters aus mehreren Maschinen. Am besten suchen Sie sich für die Experimente einen Cloudprovider und ordern dort drei kleine virtuelle Linux-VMs zum Stundentarif – für diesen Artikel kommt Ubuntu Server 20.04 LTS zum Einsatz. Trotz gestiegener Energiekosten bekommen Sie für unter 20 Euro im Monat (bei Dauerbetrieb) eine Testumgebung mit drei Maschinen. Auch wenn Sie schon absehen können, dass Sie sich beruflich niemals mit der Installation und dem Betrieb eines Kubernetes-Clusters beschäftigen müssen, weil Ihr Unternehmen ein Managed-Kubernetes-Produkt eines Providers nutzt, ist es fürs Verständnis ungemein hilfreich, mal selbst einen Cluster gebaut zu haben.

Drei Server, ein Cluster

Entstehen soll im Folgenden ein Zusammenschluss aus mehreren Servern, die am selben Ziel arbeiten: Ihre containerisierte Anwendung stabil, skalierbar und redundant auszuführen. Aber warum gleich drei Maschinen, ein Testcluster könnte man doch auch mit zwei Maschinen günstiger simulieren – oder nicht? Die Zahl drei werden Sie in der Kubernetes-Welt noch öfter lesen und dafür gibt es einen guten Grund: Kubernetes nutzt (in den allermeisten Fällen) die Key-Value-Datenbank etcd für die Verwaltung des Cluster-Zustands. Diese Datenbank kann redundant über mehrere Maschinen verteilt laufen und setzt auf den Konsens-Algorithmus Raft – und der wiederum funktioniert am besten mit einer ungeraden Anzahl Maschinen. Warum das so ist und was das mit Demokratie unter Servern zu tun hat, lesen Sie im Artikel „Verteilte Systeme mit Raft-Algorithmus“ auf Seite 98.

Ihr erster Cluster soll gleich ausfallsicher arbeiten. Daher bekommen alle drei Maschinen (in der Kubernetes-Welt Nodes genannt) die Master-Rolle und eine etcd-Kopie. Das versetzt Sie in die Lage, dass Sie die Server (etwa bei Updates) problemlos nacheinander neu starten können. Den Ausfall einer Maschine steckt der Cluster dank Raft-Algorithmus weg, die anderen sind weiter arbeitsfähig. Servern mit der Master-Rolle kommt im Cluster eine besondere Aufgabe zu: Sie stellen ein API nach innen und auf Wunsch auch nach außen bereit, über das man den Zustand des Systems abfragen und verändern kann. Nach außen nutzen alle Verwaltungswerkzeuge für Admins dieses API, nach innen steht es privilegierten Containern zur Verfügung, die darüber den Zustand von anderen Objekten und Containern erfahren und verändern können. Docker-Nutzer kennen dieses Prinzip von speziellen Containern, die den Unix-Socket /var/run/docker.sock als Volume bekommen, um andere Container zu steuern – die grafische Oberfläche Portainer ist ein weit verbreitetes Beispiel aus der Docker-Welt.

Neben diesen Master-Nodes kennt Kubernetes reine Worker-Nodes, die von dem oder den Mastern kontrolliert werden und nur Container ausführen, ohne sich mit etcd und Verwaltung zu belasten – für den Einstieg reicht es aus, die Master auch als Worker einzusetzen. In vielen produktiven Clustern laufen drei Master (das ist für etcd eine gute Größe) und beliebig viele Worker (die theoretische Obergrenze liegt bei 5000 Nodes pro Cluster).

Zum Steuern Ihrer Cluster müssen Sie sich nach der Einrichtung nicht mehr per SSH auf Ihren Server begeben, auch die Werkzeuge auf Ihrer lokalen Maschine nutzen dieses API. Und anders als der Docker-Socket ist das Kubernetes-API mit Authentifizierung ausgestattet und darf veröffentlicht werden. Damit Sie von der lokalen Maschine aus mit dem Cluster arbeiten können, brauchen Sie darauf das offizielle Kubernetes-Kommandozeilenwerkzeug `kubectl`. Wer Docker Desktop unter Windows oder macOS nutzt und den lokalen Cluster aktiviert, hat Kubectl damit direkt installiert, unter macOS bekommt man es ansonsten schnell über den Paketmanager Homebrew:

```
brew install kubernetes-cli
```

Ubuntu-Nutzer finden es über Snap:

```
sudo snap install kubectl --classic
```

Für alle anderen Betriebssysteme (mit und ohne Paketmanager) verrät die Kubernetes-Doku, wie Sie das kleine Werkzeug herunterladen, in den Programmpfad verschieben und ausführbar machen (siehe ct.de/wtzd).

Distributionskunde

Nach diesen Vorbereitungen kann die Installation des Clusters beginnen – fehlt nur noch Kubernetes selbst. Auf der offiziellen Seite kubernetes.io werden Sie aber vergeblich nach einem Download-Button fahnden. Mit Kubernetes verhält es sich wie mit dem Linux-Kernel: Den können Sie auch irgendwo aus einem schmucklosen Archiv herunterladen, werden damit aber zunächst wenig anstellen können. Wie auch Linux wollen Sie Kubernetes in Form einer Kubernetes-Distribution haben. Die bündelt all das, was zum Betrieb notwendig ist und verdrahtet die Komponenten schon mal sinnvoll. Etcd zum Beispiel wollen Sie nicht per Hand an ein nacktes Kubernetes-Binary anbinden. Kubernetes-Distributionen gibt es mittlerweile viele und ihre Wahl ist zur Wissenschaft geworden. Die meistgenutzten fallen für Selbstbetreiber schon mal raus, sie heißen GKE (Google Kubernetes Engine), AKS (Azure Kubernetes Service) und EKS (Amazon Elastic Kubernetes Service) und stecken in den Managed-Kubernetes-Angeboten der drei Branchenriesen im Cloudgeschäft.

Was Sie suchen, ist Kubernetes für „Bare-Metal-Umgebungen“. So nennt man in der Szene Installationen auf eigenen Servern, virtualisiert oder physisch. Bei der Suche nach Bare-Metal-Kubernetes werden Sie früher oder später auf das Unternehmen Rancher stoßen. Das ehemalige Start-up gehört heute zum deutschen Linux-Distributor Suse, die Rancher-Produkte funktionieren aber unabhängig von Suses Linux-Distros. Ranchers Hauptprodukt, das auch schlicht Rancher heißt, können Sie sich direkt für später merken. Es handelt sich um eine Verwaltungsoberfläche, mit der Sie Kubernetes-Cluster im eigenen Rechenzentrum oder bei den oben genannten Cloudprovidern verwalten. Rancher selbst ist ein Docker-Container, den Sie auch auf der lokalen Maschine starten können und von da aus Cluster in aller Welt hochfahren und verwalten. Der Ansatz eignet sich für Firmen, die eine Multi-Cloud-Strategie planen, ist aber überdimensioniert für den Einstieg.

Für die ersten Gehversuche (und auch für kleine und mittelgroße Kubernetes-Cluster) empfehlen wir die Distribution k3s, die ursprünglich aus dem Hause Rancher stammt und heute von der CNCF verwaltet

wird. Von dieser Organisation werden Sie auf dem Weg noch öfter hören: Die Cloud Native Computing Foundation ist eine Tochter der Linux Foundation, ihr gehört unter anderem der Open-Source-Code von Kubernetes. Außerdem verwaltet sie viele Projekte aus dem Ökosystem.

k3s ist mit dem Ziel angetreten, das Betreiben von Kubernetes-Clustern zu vereinfachen; ein paar Nischenfunktionen, die kaum jemand vermisst, sind daher rausgeflogen. Den ersten Cluster haben Sie mit k3s in wenigen Minuten einsatzbereit, weil die Distribution die Installation in ein komfortables Installationsskript verpackt hat. Öffnen Sie am besten je eine SSH-Sitzung auf Ihren drei Maschinen und platzieren Sie die Fenster für die nächsten Schritte schon einmal nebeneinander. Doch Achtung: Die nächsten Anweisungen müssen Sie zunächst nur auf einer der drei Maschinen ausführen.

Loslegen

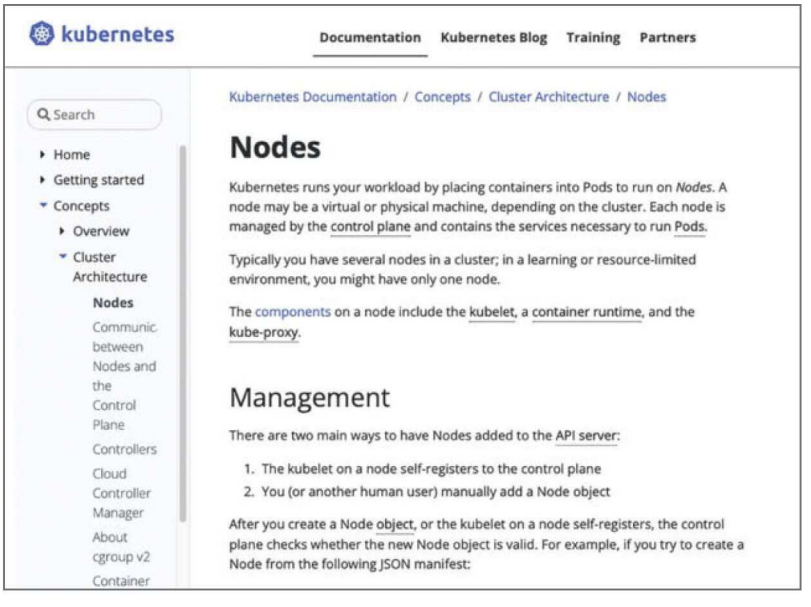
Die offizielle k3s-Anleitung zur Installation besteht aus einem Einzeiler, von dem wir ohne weitere Vorbereitungen aber abraten, weil spätere Anpassungen dadurch schwieriger werden:

```
curl -sfL https://get.k3s.io | sh -
```

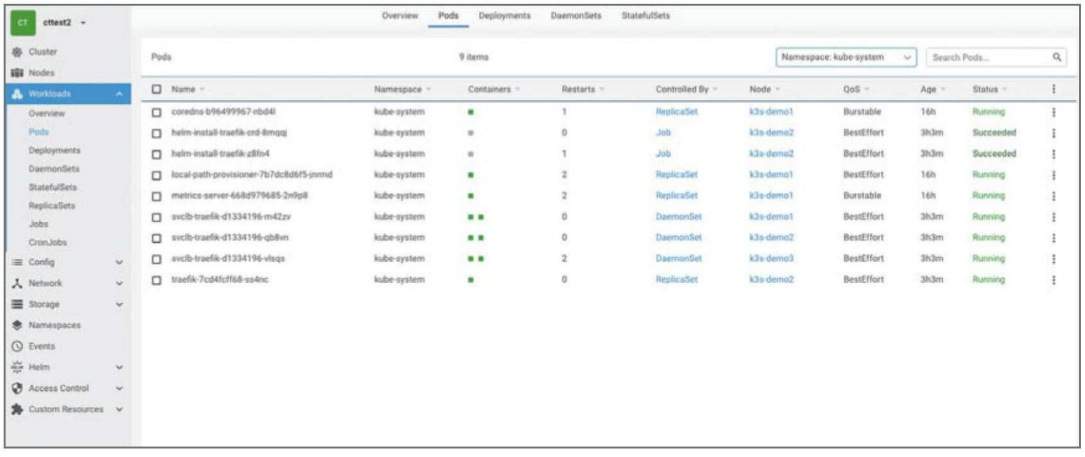
Die Zeile lädt ein Installationsskript herunter und führt es direkt aus. Wenn Sie wissen wollen, was da passiert, öffnen Sie die Adresse get.k3s.io im Browser. Das Skript lädt die k3s-Bestandteile nach und richtet einen Dienst für systemd oder OpenRC ein. Danach hat das Skript ausgedient und Kubernetes läuft rund um die Uhr im Hintergrund als Dienst mit dem Namen `k3s`, den Linux-Admins auch mit systemctl-Befehlen wie `systemctl status k3s` verwalten können.

Mit Umgebungsvariablen greifen Sie in den Einrichtungsprozess ein und legen Einstellungen fest, die dann in die Konfiguration des Dienstes gelangen. Wenn man daran nach der Installation etwas ändern will, muss man per Hand an der systemd-Konfiguration schrauben oder das Installationsskript erneut mit neuen Einstellungen drüberlaufen lassen. Sinnvoller ist es, direkt von Anfang den tiefer in der Doku versteckten Weg zu gehen und die k3s-Konfiguration in eine YAML-Datei namens /etc/rancher/k3s/config.yaml zu schreiben. Hat man später etwas daran geändert, reicht `systemctl restart k3s`, damit die Änderungen übernommen werden. Erzeugen Sie also auf dem ersten Ihrer drei Server das Verzeichnis für diese Datei:

```
mkdir -p /etc/rancher/k3s/
```



Die gut gepflegte Kubernetes-Dokumentation hat auf fast jede Frage eine Antwort. Für den Einstieg ist das aber zu umfangreich.



Kubernetes kann man nicht nur auf der Kommandozeile verwalten. Die Desktop-Anwendung Lens verbindet sich mit dem Cluster und stellt seine Details grafisch dar.

Legen Sie darin (zum Beispiel mit dem Texteditor Nano) die YAML-Datei config.yaml an:

```
nano /etc/rancher/k3s/config.yaml
```

Fürs Erste reicht darin eine Zeile:

```
disable: traefik
```

Die verhindert, dass k3s den HTTP-Proxy Traefik direkt startet, nicht weil Traefik ein schlechter HTTP-Proxy wäre, sondern damit Sie später selbst lernen, Traefik manuell zu installieren. Auf dem ersten Server ist damit alles bereit für die Installation von k3s:

```
curl -sfL https://get.k3s.io | sh -s - server↵
↳ --cluster-init
```

Die Leerzeichen in diesem Befehl sehen vielleicht falsch aus, haben aber ihre Richtigkeit. Das Installationsskript wird an `sh` übergeben und mit dem Befehl `server --cluster-init` ausgeführt. Der letzte Parameter führt dazu, dass k3s eine frische etcd-Instanz einrichtet. Nach einer Minute ist die Einrichtung erledigt und Ihr Cluster läuft. Glauben Sie nicht? Ihr erster Befehl mit `kubectl` auf dem Server selbst beweist es:

```
kubectl get nodes
```

Wenn Kubectl einen Zertifikatsfehler präsentiert, geben Sie der Installation noch ein bisschen Zeit. Sobald „Ready“ in der Tabelle erscheint, ist der Single-Node-Cluster hochgefahren. Sollte der Befehl mit einer Fehlermeldung scheitern, sind Sie nicht Root auf dem Server – k3s schränkt die Rechte auf die Konfigurationsdatei stark ein (was man mit der Zeile `write-kubeconfig-mode: "064"` in der Konfiguration ändern könnte). Mit einem vorangestellten `sudo` können Sie zugreifen. Wie schon erwähnt: Im Alltag greifen Sie selten über eine SSH-Sitzung vom Server selbst zu, sondern per `kubectl` vom heimischen Arbeitsplatz.

Bevor Sie Kubectl lokal einrichten, soll der Cluster aber um zwei weitere Mitglieder erweitert werden. Erzeugen Sie die oben angelegte Konfigurationsdatei auf den anderen beiden Maschinen. Damit auch die anderen Server als Master in den Cluster aufgenommen werden dürfen, brauchen Sie ein Token, das das k3s auf der ersten Maschine automatisch angelegt und in eine Datei geschrieben hat. Sie finden es mit folgendem Befehl:

```
cat /var/lib/rancher/k3s/server/token
```

Kopieren Sie die komplette zurückgegebene Zeichenkette in die Zwischenablage. Nun brauchen Sie nur noch die IP-Adresse des ersten Servers, der schon ein Cluster eröffnet hat. Im besten Fall können sich die drei Clustermitglieder über eine interne IP-Adresse erreichen, zur Not klappt es für die Experimentierumgebung auch mit den externen Adressen (für ein produktives System müssen Sie da später noch mal nachbessern). Fügen Sie Token und IP-Adresse in den folgenden Befehl ein und setzen Sie diesen auf den Servern zwei und drei ab:

```
curl -sfL https://get.k3s.io | ↵
↳K3S_TOKEN=<Token> sh -s - server ↵
↳--server https://<IP Server 1>:6443
```

Der Befehl setzt voraus, dass sich die Server untereinander über Port 6443 erreichen, außerdem braucht etcd die TCP-Ports 2379 und 2380. Die Ports müssten Sie in Ihren Firewalls öffnen – gute Gründe für ein internes Netz. Am Ende der Zeremonie sollte `kubectl get nodes` (auf einem der Server abgesetzt) drei gesunde Master-Nodes anzeigen.

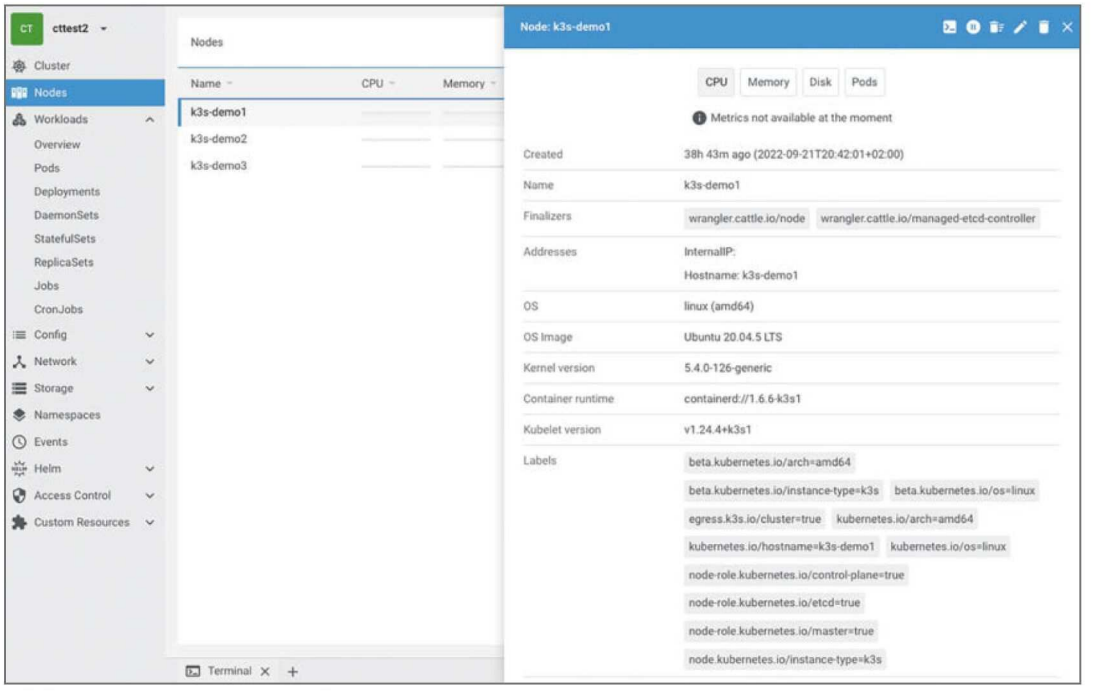
Anschließend können Sie die Fernsteuerung auf Ihrer lokalen Maschine einrichten. Dafür müssen Sie insgesamt vier Werte hinterlegen: die externe Adresse eines Ihrer Kubernetes-Master, dessen TLS-Zertifikatsinformationen sowie den öffentlichen und den privaten Schlüssel für den Benutzeraccount im Cluster. Die externe IP-Adresse kennen Sie, die anderen Informationen liegen auf allen drei Master-Servern. Mit folgendem Befehl bekommen Sie diese zu sehen:

```
cat /etc/rancher/k3s/k3s.yaml
```

Der Inhalt der Datei sieht auf den ersten Blick kompliziert aus und ist es aus gutem Grund auch: Kubectl ist dafür konzipiert, mit mehreren Clustern zu arbeiten – die meisten Nutzer haben mindestens ein Produktiv- und ein Entwicklungscluster oder gar Cluster bei mehreren Kunden, daher kann man zwischen Kontexten wechseln (und muss bei schreibenden Befehlen immer sicherstellen, dass man im richtigen Kontext unterwegs ist). Ein Kontext ist immer eine Kombination aus einem Cluster und einem Benutzer mit seinen Zugangsdaten. Verwaltet werden diese entweder alle in einer YAML-Datei oder in mehreren Dateien, wir stellen hier die Strategie mit einer Datei vor, andere Herangehensweisen beschreibt die Doku (zu finden über ct.de/wtzd).

Sofern Sie Kubectl über Docker Desktop bekommen haben, liegt in Ihrem Benutzerverzeichnis bereits der Ordner .kube, unter Linux und macOS also unter ~/.kube, unter Windows in %USERPROFILE%\.kube. Weil der Ordnername mit einem Punkt beginnt, blenden ihn viele grafische Dateiexplorer aus, über die Kommandozeile finden Sie ihn aber. Gibt es den Ordner noch nicht, weil Sie Kubectl per Hand installiert haben, legen Sie ihn zunächst an.

Kubectl erwartet in diesem Ordner eine Datei namens config (ohne Endung). Gibt es sie noch nicht, legen Sie sie an und kopieren den kompletten Inhalt



Nicht nur Einsteiger finden in der grafischen Oberfläche Lens wichtige Informationen, die auf der Kommandozeile schnell untergehen.

der Datei k3s.yaml vom Server hinein. Ändern müssen Sie dann nur die IP-Adresse 127.0.0.1:6443 durch die externe IP-Adresse eines Servers (oder durch einen DNS-Namen) mit dem Port 6443 am Ende. Geben Sie dem Kontext in Zeile 11 noch einen sprechenderen Namen als `default` geben – zum Beispiel `dev-k3s`. Anschließend sind Sie einsatzbereit.

Gibt es die Datei bereits, hat Docker sie angelegt und mit den Werten für die lokale Umgebung befüllt. Dann müssen Sie die Abschnitte vom Server einzeln in die Datei kopieren (am besten mit einem grafischen Texteditor). Zunächst die Clusterinformationen (mit angepasster IP-Adresse). Aus dem Clusternamen `default` machen Sie einen sprechenden Namen wie `dev-k3s`. Dann den Abschnitt für den User, dessen Namen Sie ebenfalls von `default` in `dev-k3s` ändern sollten. Als dritten Schritt fügen Sie einen Block unter `contexts` hinzu und kombinieren nach dem schon angelegten Schema das Cluster `dev-k3s` mit dem gleichnamigen Benutzer zu einem Kontext mit ebendiesem Namen. Wenn Sie im YAML-Salat den Überblick verloren haben, finden Sie ein Beispiel (ohne gültige Zugangsdaten) über ct.de/wtzd.

Weisen Sie Kubectl jetzt an, den konfigurierten Kontext zu nutzen:

```
kubectl config use-context dev-k3s
```

Der schon bekannte Befehl `kubectl get nodes` sollte jetzt auch aus der Ferne die drei gesunden Nodes anzeigen. Wenn nicht, könnte eine Firewall Port 6443 auf Ihrem Node blockieren. Sollten Sie Fehler beim Zusammenbau der YAML-Datei gemacht haben, beschwert sich der YAML-Parser von Kubectl mit einer hilfreichen Fehlermeldung.

Für Docker-Desktop-Nutzer gibt es noch einen zweiten Weg, den Kontext zu wechseln: Mit einem Klick auf das Wal-Logo in der Taskleiste öffnen sie ein Menü, das den Punkt Kubernetes enthält. Dahinter verbergen sich alle erkannten Kontexte für den schnellen Wechsel per Mausklick. Und noch eine Abkürzung ist dringend empfohlen: Während Ihrer nächsten Kubernetes-Lernschritte werden Sie `kubectl` oft eingeben. Kubernetes-Intensivnutzer sparen sich viel Tipperei, wenn sie sich einen Alias wie `k` dafür anlegen. `kubectl get nodes` legen Kubernetes-Profis gern auf den Alias `kgn`.

Neben Kubectl raten wir Ihnen zu einem weiteren Werkzeug, mit dem Sie auf Ihre Cluster zugreifen können: Lens (k8slens.dev) ist eine grafische Oberfläche, die als Anwendung auf Ihrer Maschine läuft und die Kontexte aus der Kubectl-Konfigurationsdatei übernimmt. Die Software ist kostenlos, schnell eingerichtet und läuft unter Windows, Linux und macOS. Welche Details Lens zeigen kann, sehen Sie im Bild auf Seite 44.

Aufbauen und abreißen

Der folgende Tipp mag etwas befremdlich wirken, zahlt sich aber später aus: Wenn Ihr Cluster läuft, sollten Sie ihn möglichst bald wieder abreißen und den Bau, soweit es geht, automatisieren – das ist eine Herangehensweise, an die man sich im Cloud-Native-Umfeld früh gewöhnen sollte. Erzeugen Sie Ihre Infrastruktur immer reproduzierbar und schaffen Sie handgeknüpfte Strukturen ab. Zum restlosen Entfernen von k3s gibt es den Befehl

```
/usr/local/bin/k3s-uninstall.sh
```

Für eine reproduzierbare Installation verpacken Sie alle Schritte im einfachsten Fall in Bash-Skripte; wenn Sie mit Werkzeugen wie Ansible und Terraform vertraut sind, nutzen Sie diese für die Einrichtung von Servern, Firewalls, DNS-Einträgen und schließlich k3s.

Zum Schluss

Glückwunsch, Sie sind jetzt Betreiber eines selbst gebauten Clusters mit echter Redundanz dank verteilter Master-Rolle. Fahren Sie zum Test mal einen der Server herunter und testen, ob `kubectl get nodes` auf den anderen beiden noch funktioniert. Im Cluster läuft aber (abgesehen von ein paar System-Containern) noch nichts. Einen ersten Container, der eine einfache Website auf Port 30.000 veröffentlicht, haben Sie schnell in Betrieb: Über ct.de/wtzd finden Sie eine YAML-Datei namens first-pod.yml zum Download und fürs Selbststudium. Laden Sie diese auf Ihre lokale Maschine mit installiertem Kubectl, navigieren Sie auf der Kommandozeile in den Ordner mit der Datei und installieren die Zusammenstellung im Cluster:

```
kubectl -f first-pod.yml apply
```

Wenig später sollten alle drei Maschinen auf ihren externen IP-Adressen auf Port 30.000 mit einer Website antworten. Im folgenden Artikel erfahren Sie, warum Container in Kubernetes in sogenannten Pods stecken, wie Sie solche erzeugen und sie mit der Außenwelt verdrahten. (jam) c't

Literatur

[1] Holger Bleich und Jan Mahn, **Wolkenangebotsvielfalt**, Sechs europäische Cloudprovider im Überblick, c't 21/2021, S. 62

Dokumentation und Beispiele

ct.de/wtzd

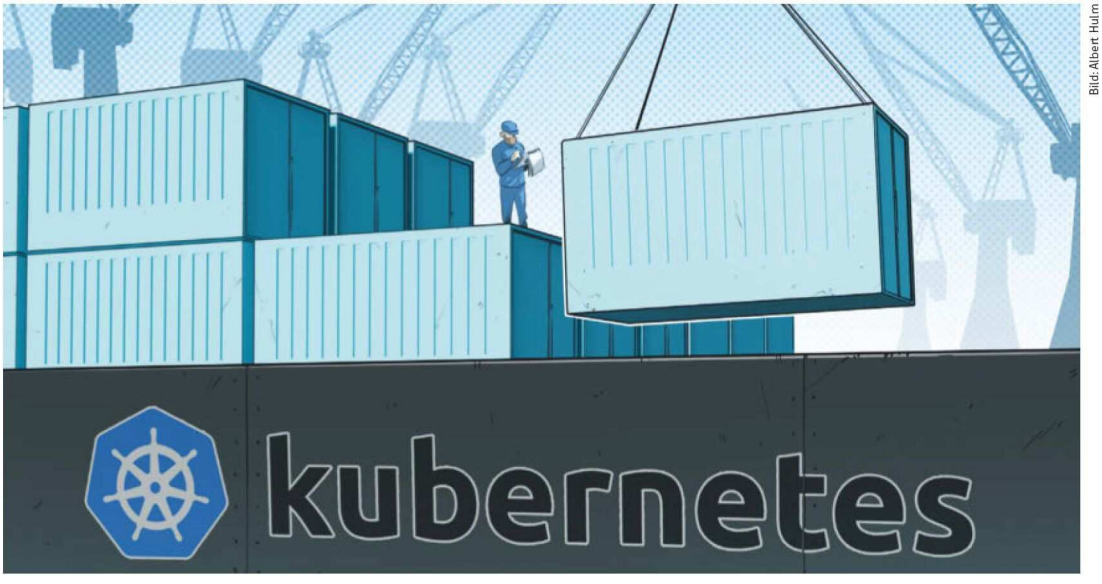


Bild: Albert Hulm

Container, Pods und Deployments

Im zweiten Teil der Reihe füllen Sie Ihren Kubernetes-Cluster mit Containern, spielen Updates ohne Downtime aus und lernen verschiedene Administrationsstile kennen.

Von **Jan Mahn**

Zum Auftakt dieser Reihe haben Sie im vorigen Artikel erfahren, wie aus drei Linux-Maschinen ein Kubernetes-Cluster wird. Einen solchen brauchen Sie für diesen zweiten Teil, ob nun selbstgebaut, im Rechenzentrum Ihres Unternehmens oder bei einem Provider fix und fertig gemietet. Auf Ihrer lokalen Maschine sollte das Kommandozeilenwerkzeug Kubectl installiert und mit den Zugangsdaten des Clusters versorgt sein.

Der erste Teil endete mit einem kleinen Erfolgserlebnis: Ihr erster Kubernetes-Container lief und zeigte eine Beispielseite auf Port 30000. Die Definition haben Sie aus unserem Beispiel heruntergeladen und in Ihren Cluster gebracht. In diesem zweiten Teil erfahren Sie, wie die YAML-Definitionen funktionieren. Damit Sie wieder mit einem leeren Cluster starten, müssen Sie zunächst aufräumen, wenn Sie das Beispiel bereits im Cluster laufen haben:

```
kubectl delete pod my-first-nginx
kubectl delete service my-service
```

Damit Sie nichts abtippen müssen (was bei YAML immer fehlerträchtig ist), finden Sie alle YAML-Schnipsel der Anleitung zum Download über ct.de/wfjs.

Hülsenfrüchte

In der Docker-Welt sind Container die kleinste Einheit, die man starten, stoppen und entfernen kann. Jeder Container entsteht aus einem Abbild, wird vom Rest des Betriebssystems gekapselt und mit einer virtuellen Netzwerkkarte für Kontakte mit der Außenwelt versehen. Kubernetes erweitert dieses Konzept und steckt Container immer in eine Hülle, die Pod genannt wird. In so einem Pod dürfen ein oder mehrere Container laufen, sie teilen sich eine gemeinsame Netzwerkverbindung und können sich auf Wunsch auch Ordner eines Dateisystems teilen. Nach außen erscheinen sie als eine Einheit und können immer nur zusammen verschoben und geklont werden.

Wie immer bei zusätzlichen Abstraktionsebenen, die Kubernetes im Vergleich zu Docker einzieht, gilt auch hier: Man muss dieses Angebot nicht nutzen und im Alltag enthalten viele Pods (wenn nicht gar die meisten) nur einen Container. Auf keinen Fall sollte man seine gesamte Anwendung (zum Beispiel Frontend, Backend und Datenbank) in einen Pod stopfen, dann könnte man es mit der Containerisierung auch ganz lassen, weil man ein unflexibles Monstrum geschaffen hätte.

Die Grundregel: Jede Komponente, die einzeln aktualisiert und skaliert werden soll, bekommt ihren eigenen Pod. Mehrere Container in einem Pod kann man zum Beispiel dann einsetzen, wenn ein Container ein kleines Helferlein in Form eines anderen Containers braucht, der einmalig oder regelmäßig eine Konfiguration einliest und in ein Pod-internes Dateisystem legt. So ein Hilfscontainer heißt in der Kubernetes-Welt Sidecar, früher oder später werden Ihnen solche Konstrukte begegnen. Ihr erster Pod kommt ohne solche Feinheiten aus. Um ihn anzulegen, brauchen Sie irgendwo auf Ihrer lokalen Maschine eine Datei (am besten in einem eigenen Ordner), die Sie zum Beispiel pod.yml nennen (auf den Namen der Datei kommt es Kubernetes nicht an). Darin folgende Zeilen:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-first-nginx
  labels:
    app: my-nginx
spec:
  containers:
    - name: nginx
      image: nginx:alpine
      ports:
        - containerPort: 80
```

Dieser Schnipsel enthält die Definition des Pods mit dem Namen `my-first-nginx`. Unterhalb von `spec:` bekommt er seine Eigenschaften zugewiesen: Die Liste der Container enthält nur einen einzigen Eintrag, erzeugt aus dem Abbild `nginx:alpine` (wie auch Docker nutzt die Kubernetes-Distribution k3s den Docker-Hub als Standard-Registry für Abbilder, Sie können aber auch jede andere Registry vor den Namen schreiben). Insgesamt werden in dieser Definition gleich mehrere Namen vergeben; der Pod selbst braucht einen nach außen einzigartigen Namen, festgelegt als Teil der `metadata`. Hier wird dem Pod auch direkt ein Label angeheftet – das kommt später zum Einsatz, um diesen Pod zu identifizieren. Der Name eines Containers (hier `nginx`) muss nur innerhalb des Pods einzigartig sein.

Navigieren Sie auf der Kommandozeile in den Ordner mit der Datei namens pod.yml und weisen Kubectl an, den Schnipsel in den Cluster zu bringen:

```
kubectl -f pod.yml apply
```

Wenn die Verbindung zum Cluster erfolgreich war, sollte Kubectl vermelden, dass ein Objekt angelegt wurde. Führen Sie den Befehl zum Test direkt noch einmal aus, meldet die Software, dass es nichts zu ändern gab. Dieselbe Datei könnten Sie jetzt von jedem anderen Rechner, auf dem die Zugangsdaten liegen, in den Cluster schieben, das Kubernetes-API würde immer prüfen, ob es einen Pod dieses Namens schon gibt und bei Bedarf dessen Attribute ändern. Um auch mal eine Änderung gesehen zu haben, ändern Sie den Namen des Images zu `nginx:1.23-alpine` und schicken den Änderungswunsch per obenstehendem Apply-Befehl in den Cluster. Einen Befehl, der wie `docker compose up` und `down` funktioniert, gibt es in Kubernetes nicht, die Änderung wird sofort umgesetzt. Um zu sehen, ob Ihr Pod wirklich läuft, brauchen Sie einen weiteren Unterbefehl von `kubectl get` (`kubectl get nodes` haben Sie bereits kennengelernt):

```
kubectl get pods
```

Auch das Entfernen von Pods ist keine schwarze Magie und die Syntax ist einleuchtend:

```
kubectl delete pod my-first-nginx
```

Nachdem Sie den letzten Befehl ausprobiert haben, bringen Sie den Pod mit dem obenstehenden Apply-Befehl einfach wieder zurück in den Cluster. Solche Operationen laufen in Kubernetes alle etwas fixer ab als das Starten und Stoppen in Docker.

Der Nginx-Container läuft jetzt, von außen sehen Sie davon aber nichts, die Webseite ist noch nirgends veröffentlicht. Auch wenn die letzten beiden Zeilen in der Definition vermuten lassen, dass hier Port 80 auf der externen Netzwerkkarte freigegeben wurde – dem ist nicht so, der Port ist bisher nur auf Container-Ebene geöffnet.

Damit ein Port auf der externen Netzwerkkarte abgehört wird, brauchen Sie ein weiteres Kubernetes-Objekt. In der Kubernetes-Welt sind alle Dinge, die man in den Cluster befördert, Objekte. Das Objekt vom Typ Pod (`kind: Pod`) haben Sie bereits kennengelernt, im zweiten Schritt brauchen Sie einen Service, also ein Objekt vom `kind: Service`. Für dessen Definition können Sie entweder eine neue Datei anlegen oder in der bestehenden Datei unterhalb der Pod-Definition mit der Zeile `---` ein weiteres Objekt einleiten. Diese beiden Optionen stehen Ihnen grundsätzlich zur Verfügung und es ist Ihre Entscheidung, wie Sie Ihre Dateien zuschneiden (anders als bei Docker-Compose). Die Definition für den Service sieht wie folgt aus:

```
---
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: default
spec:
  type: NodePort
  selector:
    app: my-nginx
  ports:
    - protocol: TCP
      nodePort: 30000
      port: 80
```

Wenn diese YAML-Epen Sie abschrecken, hier ein paar Worte zur Einordnung: Im Alltag muss man solche Konstrukte nicht auswendig in den Editor tippen. Ziemlich schnell hat man alle wesentlichen Objekte für die eigenen Anwendungsfälle zusammengebaut und kann sich ab dann an eigenem YAML bedienen. Außerdem sind alle Objekte nach demselben Schema aufgebaut, das man schnell verinnerlicht. Unter `metadata:` liegen Attribute, die das Objekt nach außen beschreiben und auffindbar machen – das braucht man immer, wenn Objekte sich auf andere Objekte beziehen. Neben dem Pflichtattribut `name` kann man hier `labels` und `annotations` zur Wiedererkennung anheften.

Unterhalb von `spec:` wird das Objekt selbst ausgestaltet. Dieser Service ist vom Typ `NodePort`, eine eher selten genutzte (für den Einstieg aber ganz nützliche) Spielart, die Ports der externen Netzwerkkarte weitergibt. Später werden Sie Services vor allem dafür einsetzen, einen Pod innerhalb des Clusters erreichbar zu machen, damit ein API-Pod zum Beispiel seine Datenbank erreichen kann.

Im `selector:` findet eine Verknüpfung von Objekten statt: Der Service soll nach einem Pod suchen, der das Kriterium `app: my-nginx` erfüllt. Gesucht wird dabei innerhalb der Labels. Wenn Sie sich die Pod-Definition noch einmal ansehen, erkennen Sie, dass genau dieses Attribut am Pod gesetzt ist.

Das ist im Vergleich zu Docker eine zusätzliche Ebene der Abstraktion – in der Docker-Compose-Datei reicht es aus, unterhalb von `ports:` einen Port der Netzwerkkarte mit einem Container zu verknüpfen, etwa wie folgt:

```
# Docker-Compose zum Vergleich
web:
  image: nginx:latest
  ports:
  -80:80
```

Sinnvoll ist die zusätzliche Komplexität durchaus: So ist es möglich, mehreren Pods das Label `app: my-nginx` zu geben. Die Netzwerkschicht in Kubernetes würde eingehende Anfragen nacheinander auf all diese verteilen. Ohne viel Arbeit haben Sie einen internen Load-Balancer. Im nächsten Abschnitt erfahren Sie, wie Sie einen Pod klonen und je eine Instanz auf jeden Ihrer Server im Cluster legen.

Vorher ist der letzte Abschnitt `ports:` im Service erklärungsbedürftig. Darin wird die Verbindung zur Außenwelt hergestellt. TCP-Verkehr, der auf den externen Netzwerkkarten auf Port 30000 ankommt, wird zu Port 80 der Pods geschickt. Kleinere Ports als 30000 kann man nicht mit einem `NodePort` belegen, gedacht ist ein solcher Service nicht für die fertige Anwendung, sondern mehr für Experimente und Admin-Hintertüren. Später lernen Sie einen besseren Weg für eingehenden Verkehr kennen.

Speichern Sie die Service-Definition ab und schieben sie mit Kubectl in den Cluster. Der sollte vermelden, dass er ein Objekt angelegt hat, danach

erreichen Sie eine Nginx-Beispielseite auf Port 30000 auf allen drei externen IP-Adressen Ihres Clusters. Dabei spielt es auch keine Rolle, auf welchem Node der Pod gestartet wurde, der Verkehr kommt dank interner Cluster-Magie immer an. Sie könnten jetzt zum Beispiel einen externen Load-Balancer vor den gesamten Cluster hängen und die Anfragen verteilen.

Um herauszufinden, auf welchem Node Ihr Pod gestartet wurde, können Sie Kubectl nutzen:

```
kubectl get pods -o wide
```

Skalieren, bitte

Für diesen einzelnen Pod hätten Sie sich den ganzen Aufwand bis hier sparen können, denn noch läuft ja nur eine Instanz. Ausfallsicher wird Ihr Konstrukt erst, wenn eine Nginx-Kopie auf jedem der drei Nodes läuft. Dafür gibt es eine weitere Abstraktionsschicht: das Deployment. Das enthält die Schablone (`template`) für einen Pod, die Kubernetes beliebig oft anwenden kann. In der folgenden YAML-Definition sehen Sie ein Deployment für den obigen Pod – und das meiste sieht vertraut aus:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-first-nginx-deployment
  labels:
    app: my-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-nginx
  template:
    metadata:
      labels:
        app: my-nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Im `metadata`-Abschnitt bekommt das Deployment wie gewohnt Name und Label, damit es selbst später auffindbar ist. Unterhalb von `spec:` dann die zentrale Information im Attribut `replicas`: Drei Kopien sollen entstehen. Es folgt ein `selector`, der wie der eines Service funktioniert und nach einem anderen Objekt sucht. In diesem speziellen Fall muss man aber innerhalb desselben Objekts dafür sorgen, dass dieses Suchkriterium erfüllt wird: Unterhalb von `template:` folgt die Blaupause für die Pods, die alle das geforderte Label `app: my-nginx` angeheftet bekommen.

Um jetzt vom Einzel-Pod auf das Deployment umzustellen, löschen Sie zuerst den Pod:

```
kubectl delete pod my-first-nginx
```

Entfernen Sie seine Definition aus der YAML-Datei und ersetzen ihn durch den obigen Deployment-Abschnitt, die Definition des Service bleibt unberührt. Dann schieben Sie die Änderungen in den Cluster. Der Befehl `kubectl get pods -o wide` zeigt den Erfolg: Es liegen drei Pods mit einer Zufallszeichenkette im Namen im Cluster. Im Browser sehen Sie von der Redundanz noch nichts, weil alle Nginx-Container dieselbe Seite ausliefern – die Anfragen werden aber schon auf alle drei Pods verteilt. Glauben Sie nicht? Dann ändern Sie das Image zu:

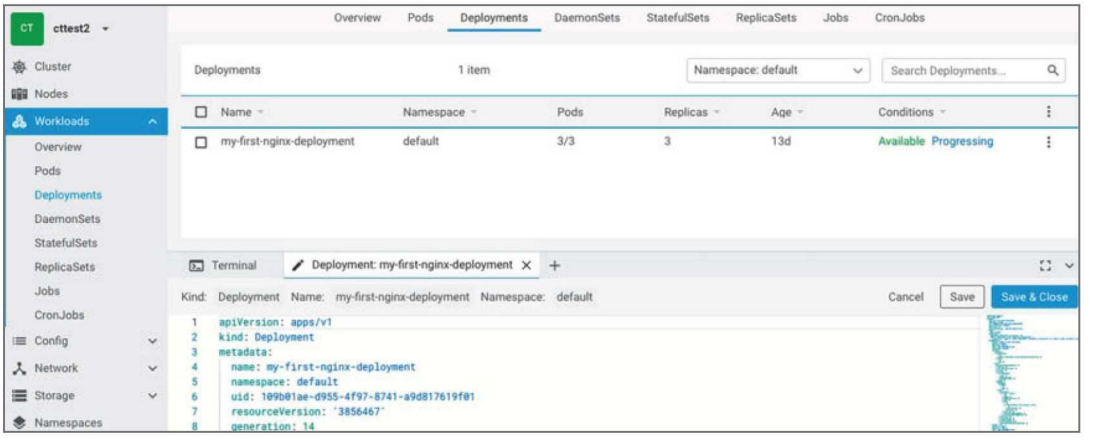
```
image: containous/whoami
```

Dieser Container wurde genau für solche Tests erfunden und zeigt in der ersten Zeile der Ausgabe den Namen seines Pods an. Mit dem Apply-Befehl tauschen Sie das Abbild aus — dann müssen Sie ihrem Browser nur noch das Cachen abgewöhnen (oder Curl benutzen) und mehrere Anfragen absetzen, um die Lastverteilung des Kubernetes-Service bei der Arbeit zu sehen. Sie sollten nacheinander Antworten Ihrer drei Pods bekommen. Wenn Ihnen drei Pods nicht mehr reichen, gibt es gleich zwei Wege, das Deployment zu skalieren. Entweder ändern Sie Zeile `replicas: 3` oder Sie setzen einen eigenen Befehl dafür ab:

```
kubectl scale deployment/↵
↳my-first-nginx-deployment ↵
↳--replicas=5
```

Letztere Strategie hat aber ihre Nachteile: Wenn Sie oder ein Admin-Kollege später die YAML-Datei per Apply-Befehl ins Cluster schieben, wird die Änderung überschrieben und wieder der Wert für `replicas` aus der Datei benutzt.

Ein Deployment ist verhältnismäßig hartnäckig. Kubernetes wird sich immer wieder darum kümmern, dass die geforderte Anzahl Pods existiert. Das können Sie ausprobieren, indem Sie sich die exis-



Jedes Objekt, das man aus einer YAML-Definition in den Cluster bringt, reichert Kubernetes mit weiteren Attributen an. Um die anzuzeigen, greift man zu Kubectl auf der Kommandozeile oder einem grafischen Werkzeug wie Lens.

tierenden Pods anzeigen lassen, einen Namen herauskopieren und diesen mit `kubectl delete pod` entfernen. Schneller als Sie schauen können, hat Kubernetes den Bestand wieder aufgefüllt.

Der Reihe nach

Mit Deployments kennen Sie jetzt eines der mächtigsten Werkzeuge für reibungsarme Softwareverteilung. Aber noch ist Ihr erstes Deployment nicht perfekt eingerichtet. Im Idealfall sollten Sie in der Lage sein, ein neues Abbild in den Cluster zu bringen, ohne dass die Nutzer einen Ausfall bemerken. Keine einzige HTTP-Anfrage darf verloren gehen. Die Zeiten, in denen Sie Ihren Nutzern ein längeres Wartungsfenster ankündigen müssen, weil Sie ein Update ausrollen, sind damit vorbei.

Bisher klappt das aber nicht: Wenn Sie in der aktuellen Konfiguration den Namen des Images wechseln und die Änderung übernehmen, wird Kubernetes alle alten Pods abreißen und neue Pods hinstellen. Weil Nginx sehr schnell einsatzbereit ist, kann es sein, dass man keinen Ausfall bemerkt – aber die wenigsten Container sind so schnell startklar wie ein nackter Nginx, manch ein Container braucht Minuten, bis er sich berappelt hat. Was Sie in den meisten Deployments nutzen wollen, ist das `RollingUpdate`. Ist das aktiv, wird Kubernetes die neuen Pods nacheinander hoch- und die alten runterfahren, sodass jederzeit eine definierte Anzahl Pods einsatzbereit ist. Die Funktion aktivieren Sie mit folgendem Schnipsel unterhalb von `spec:` des Deployments (nicht des Templates):

```
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 50%
      maxSurge: 1
```

Die letzten beiden Angaben sind optional: `maxUnavailable` gibt an, wie viele Pods parallel im nicht einsatzbereiten Zustand sein dürfen. `maxSurge` legt fest, wie viele Pods über `replicas` hinaus existieren dürfen. Das ist erst dann nötig, wenn man ressourcenhungrige Pods nutzt und verhindern muss, dass zu viele davon Prozessor und RAM überbelasten. Die Angaben für beide Einstellungen dürfen absolut oder relativ sein.

Mit der oben beschriebenen Definition räumt Kubernetes maximal 50 Prozent der Pods weg, rundet aber immer ab, sodass von dreien nur ein Pod verschwindet. Dann legt das System zwei neue Pods an (bis zu vier gleichzeitige sind durch `maxSurge` ja erlaubt). Um dieses Schauspiel zu sehen, bauen Sie zunächst den Schnipsel in Ihr Deployment ein und ändern das Image (zum Beispiel wieder auf `nginx:alpine`). Bevor Sie die Änderungen anwenden, öffnen Sie ein zweites Kommandozeilenfenster und führen darin folgenden Befehl aus:

```
kubectl get pods -w
```

Der Parameter `-w` aktiviert den Watch-Modus, Sie sehen Änderungen damit in Echtzeit. Wenden Sie

dann im anderen Fenster die Änderungen an. Im Schnelldurchlauf sehen Sie, wie Kubernetes Pods auf- und abbaut.

Perfekt ist der fliegende Wechsel aber noch nicht. Die Pods werden gestartet und sofort für einsatzbereit betrachtet – für Perfektion müssen Sie zwei weitere Konzepte kennenlernen: LivenessProbes und ReadinessProbes. Mit diesen kann Kubernetes durch regelmäßige Prüfung herausfinden, ob ein Pod einsatzbereit ist. Die beiden Funktionen arbeiten gut im Zusammenspiel, gehören aber zu den am häufigsten missverstandenen Kubernetes-Funktionen. Beide werden unterhalb von `spec:` an einem Container definiert und beschreiben einen Test, der wiederholt ausgeführt wird. Das kann ein Kommandozeilenbefehl im Container selbst sein. Bei allen Containern, die irgendwie übers Netzwerk erreichbar sind, kann man einen TCP-Port anfragen lassen. Container, die HTTP anbieten, prüft man mit einer spezialisierten HTTP-Prüfung.

Zunächst zur ReadinessProbe. Ihre Aufgabe ist festzustellen, ob ein Container bereit ist, Anfragen anzunehmen. Das Ergebnis dieser Prüfung nutzt ein vorgeschalteter Service bei der Entscheidung, welchem Pod er Anfragen zustellt. Schlägt die ReadinessProbe fehl, bekommt der Pod solange keine Anfragen, bis sie erfolgreich ist. Ein neu gestarteter Pod muss sich immer erst beweisen, bevor er Verkehr zugestellt bekommt. In Kombination mit `rollingUpdate` stellt das sicher, dass wirklich alle Anfragen ankommen:

```
spec:
  containers:
    - name: nginx
      image: nginx:alpine
      ports:
        - containerPort: 80
      readinessProbe:
        httpGet:
          path: /index.html
          port: 80
        periodSeconds: 10
```

Alle 10 Sekunden wird Kubernetes mit dieser Konfiguration versuchen, die Seite index.html zu öffnen. Kommt ein Statuscode zwischen 200 und 299 (in diesem Fall 200) zurück, wird der Container als empfangsbereit eingestuft und mit Anfragen von außen belästigt. In einer echten Anwendung sollten Sie nicht die Startseite index.html für die Prüfung einsetzen – die kann ja sehr groß werden. Bauen Sie lieber einen eigenen Endpunkt wie /ready für diesen Zweck, der nur „ok“ zurückgibt. Viele fertige Container sind bereits für den Betrieb mit LivenessProbes vorbereitet und in der Dokumentation findet sich der passende Pfad. Wenn Sie wissen, dass ein Container überdurchschnittlich lange beim Start braucht, können Sie die Ausführung der ersten LivenessProbe mit der Angabe `initialDelaySeconds` (auf gleicher Höhe mit `periodSeconds`) auch verzögern.

Die zweite Prüfung braucht man eher in seltenen Fällen: Die LivenessProbe kann Container aufspüren, die sich in einen Status manövriert haben, aus dem sie aus eigener Kraft nicht mehr herauskommen. Ganz selten kann es zum Beispiel mal vorkommen, dass eine Software zwar läuft, aber keine Aufträge mehr verarbeitet. Gibt es einen Endpunkt, der dann einen Fehler zurückgibt, bemerkt das eine LivenessProbe, Kubernetes löscht den Pod und ersetzt ihn. In der YAML-Datei würden Sie eine LivenessProbe wie eine ReadinessProbe definieren. Für den Einstieg brauchen Sie aber keine LivenessProbe, schon gar nicht für einen statischen Webserver wie Nginx. In echten Anwendungen, die Sie selbst programmieren, lohnt es sich aber, etwas Denkarbeit in die Gestaltung eines geeigneten Endpunkts wie /live zu stecken.

Aber Achtung: Niemals, wirklich niemals darf eine LivenessProbe so konstruiert sein, dass sie von einem anderen Pod abhängig ist – damit haben sich vor Ihnen schon andere sehr unschöne Domino-Effekte gebaut. Stellen Sie sich vor, Sie haben eine Datenbank im Cluster installiert. Außerdem diverse Pods, die auf diese Datenbank zugreifen und zum Beispiel ein API anbieten. Für die LivenessProbes haben Sie eigens den Endpunkt /live gebaut, der zum Test etwas aus der Datenbank holt und im Erfolgsfall „ok“ meldet. Fällt jetzt die Datenbank kurz aus, werden schlagartig alle Pods in den Fehlerzustand wechseln, weil ihre LivenessProbe scheitert. Kubernetes wird sie alle ausmustern und neue Pods starten. Im Kleinen geht das fix, im großen Cluster richtet das minutenlanges Chaos an und die Anwendung wird länger unerreichbar. Wenn Sie eigene Endpunkte für LivenessProbes programmieren, achten Sie immer darauf, dass der Endpunkt wirklich nur dann einen Fehler zurückgibt, wenn ein Neustart des Pods das Problem lösen kann.

Wenn Sie das Wissen aus diesem Teil anwenden wollen und eine Übungsaufgabe suchen, versuchen Sie doch einmal, eine containerisierte Anwendung, die Sie aktuell mit Docker oder Podman nutzen, in einen Pod zu verpacken und mit einer LivenessProbe auszustatten. (jam) c't

Dokumentation und Beispiele

ct.de/wfjs

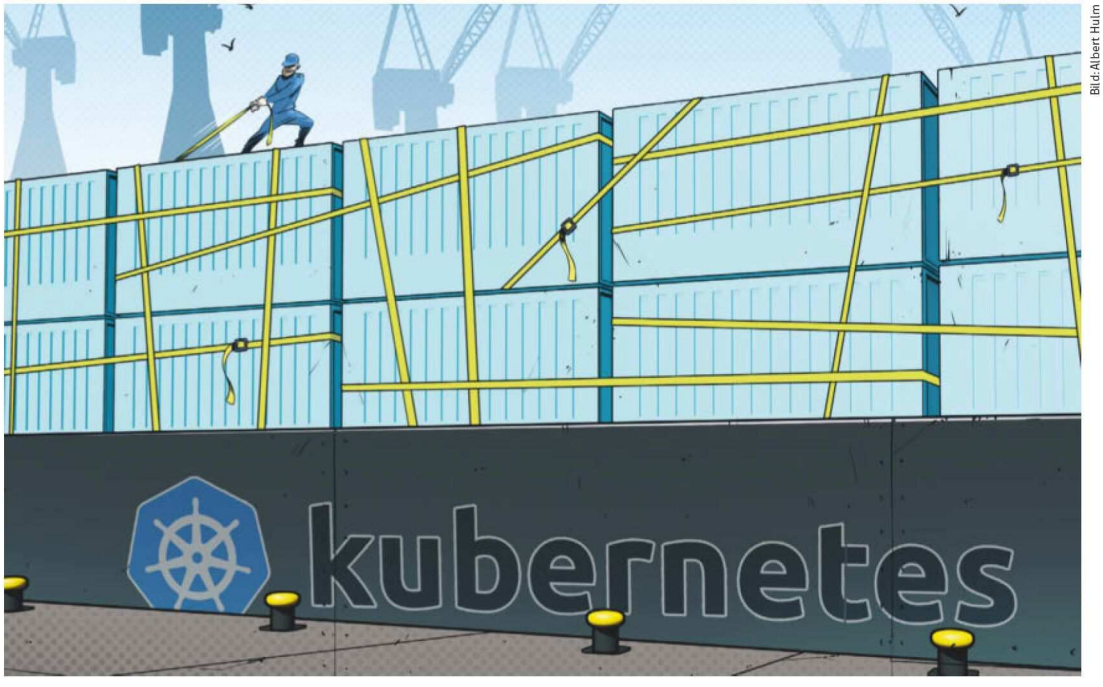


Bild: Albert Hulm

Services und Ingress mit Traefik

Container kommen selten allein vor und brauchen meist Kontakt zu anderen sowie zur Außenwelt. Im dritten Teil der Kubernetes-Reihe vernetzen Sie Container per Kubernetes-Service und installieren den Reverse-Proxy Traefik bequem mit dem Paketmanager Helm. Dann ist Ihr Cluster bereit für Anfragen von außen.

Von **Jan Mahn**

Pro Container darf nur ein Prozess laufen – diese Grundregel lernen Container-Einsteiger als erste Lektion. Will man wirklich von Containerisierung profitieren, muss man beim Planen von Images der Versuchung widerstehen, alle Komponenten in einen Container zu stecken. Datenbank und Anwendung zum Beispiel gehören in separate Container, sollen separat vervielfältigt und aktualisiert werden. Damit sie zusammenarbeiten können, muss man sie so vernetzen, dass sie sich gegenseitig finden und Informationen austauschen können.

Was in einfachen Containerumgebungen wie Docker und Podman fast von allein passiert, kann und muss man in der Kubernetes-Welt konfigurieren. Im

vorhergehenden Artikel „Container, Pods und Deployments“ ab Seite 46 haben Sie das Konzept des Service kennengelernt, der Anfragen entgegennimmt und an Pods weiterleitet, die mit einem bestimmten Label versehen sind. Ganz automatisch arbeitet ein Service als Load-Balancer und kümmert sich ebenfalls darum, nur solche Pods mit Anfragen zu belästigen, die von einer LivenessProbe für empfangsbereit erklärt wurden.

Genutzt haben Sie Services in den ersten Teilen dieser Reihe, um Anfragen von außen an Pods durchzuleiten – aber genauso braucht man sie, damit sich Pods untereinander zuverlässig finden. Das Paradebeispiel: Ein Pod, der ein API bereitstellt, soll den Pod mit seiner Datenbank erreichen. Grundsätzlich würde das in Kubernetes auch ohne Service funktionieren, weil jeder Pod im Cluster eine IP-Adresse und einen DNS-Eintrag bekommt. Wie bei Docker gilt: Arbeiten Sie niemals mit internen IP-Adressen, die wechseln immer wieder und sind nicht berechenbar. Machen Sie stattdessen von der Kubernetes-Namensauflösung Gebrauch und arbeiten Sie immer mit DNS-Namen.

Den Namen eines Pods direkt in der Konfiguration eines anderen Pods zu hinterlegen ist durchaus möglich, aber keine gute Idee: Sobald Sie skalieren wollen und mehrere identische Pods haben, brauchen Sie dazwischen einen Service als Vermittler. Wenn Sie, wie bereits gezeigt, ein Deployment einsetzen, das Pods aus einer Schablone erzeugt, bekommen diese je einen Namen mit einer Zufallskomponente – auf die Pod-Namen können Sie dann ebenso wenig vertrauen wie auf interne IP-Adressen. Es lohnt sich also, von Anfang an mit Services zu arbeiten, wenn sich Container untereinander ansprechen sollen.

Mit Namespace

Kubernetes stellt intern einen DNS-Server bereit, den alle Pods standardmäßig nutzen. Um die DNS-Namensauflösung innerhalb eines Clusters zu verstehen, müssen Sie sich zunächst mit einem anderen Konzept vertraut machen: dem Namespace. Im Cluster liegen die meisten Objekte in einem Namespace – wenn man beim Anlegen keinen explizit angibt, ist das immer der Namespace `default`. Mit Namespaces schafft man Ordnung im Cluster und trennt Bereiche organisatorisch. In Mehr-Admin-Umgebungen dienen sie auch der Berechtigungsverwaltung: Weil das Kubernetes-API ein umfangreiches Berechtigungssystem mitbringt, kann man Frontend-Entwicklern zum Beispiel explizit Schreibrechte auf ihren Namespace `frontend` zuweisen, im Namespace `backend` brauchen sie nur Lese- oder gar keine Rechte. Wenn Sie sich im Detail für Berechtigungsverwaltung interessieren, finden Sie Informationen in der Dokumentation über ct.de/wwhr.

Einen neuen Namespace haben Sie schnell angelegt, es handelt sich um ein gewöhnliches Objekt (wie Pods oder Services) mit den üblichen Spielregeln. Es reicht also, eine YAML-Datei mit wenigen Zeilen anzulegen und per Kubectl in den Cluster zu bringen:

```
apiVersion: v1
kind: Namespace
metadata:
  name: backend
```

Nun hat Ihr Cluster den Namespace `backend`. Um zu sehen, welche Pods darin existieren, müssen Sie einen gewohnten Befehl erweitern:

```
kubectl get pods -n backend
```

Haben Sie mal vergessen, in welchem Namespace ein Objekt liegt, können Sie auch Objekte in allen Namespaces anzeigen:

```
kubectl get pods --all-namespaces
```

Diese Parameter funktionieren auch mit anderen Objekten, zum Beispiel mit `kubectl get services`, auch Services liegen in einem Namespace. Und wenn Sie sich länger in einem bestimmten Namespace umsehen wollen, können Sie den Kubectl-Kontext anpassen und den Parameter `-n` fortan weglassen:

```
kubectl config set-context --current↵
↳ --namespace=backend
```

Der Namespace ist Teil des vollständigen Hostnames eines Service. Ein Service namens `my-service`, der im Namespace `default` liegt, heißt mit vollem Namen `my-service.default.svc.cluster.local`. Wer im Cluster den Kubernetes-internen DNS-Server nach der zugehörigen IP-Adresse befragt, bekommt eine Antwort. Ganz so lang muss die Anfrage aber nicht sein. Die Endung `.svc.cluster.local` wird als Standard-Domain angenommen und kann weggelassen werden. Sucht ein Pod einen Service im selben Namespace, kann er auch diesen weglassen. Namespace-intern

WordPress-Backend

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: backend
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wb-database-deployment
  namespace: backend
  labels:
    app: wp-database
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: wp-database
  template:
    metadata:
      name: wp-database
      namespace: backend
      labels:
        app: wp-database
    spec:
      containers:
        - name: mariadb
          image: mariadb:latest
          ports:
            - containerPort: 3306
          env:
            - name: MARIADB_ROOT_PASSWORD
              value: "verySecret"
            - name: MARIADB_DATABASE
              value: "wp"
            - name: MARIADB_USER
              value: "wp"
            - name: MARIADB_PASSWORD
              value: "secretWp"
---
apiVersion: v1
kind: Service
metadata:
  name: wp-database
  namespace: backend
spec:
  selector:
    app: wp-database
  ports:
    - protocol: TCP
      port: 3306
      targetPort: 3306
```

würde auch eine Anfrage für `my-service` zum Erfolg führen.

Am Beispiel

In einem typischen Szenario mit einer echten Anwendung wird das Konzept schnell deutlich: Die Blog-Software WordPress bietet sich als Beispiel an. Sie besteht aus einem Container mit der Anwendung und einem Container mit der Datenbank MariaDB. Dafür brauchen Sie mehrere Kubernetes-Objekte. Deren Definition finden Sie in den Kästen „WordPress-Backend“ auf S. 54 und „WordPress-Frontend“ auf S. 55, Frontend und Backend getrennt. Die YAML-Definition haben wir Ihnen über ct.de/wwhr zum Download bereitgestellt. Laden Sie diese Datei herunter und bringen die Definition per Kubectl in Ihren Cluster.

Nach wenigen Sekunden läuft eine fertige WordPress-Instanz, die Sie über die externen IP-Adressen Ihres Clusters auf Port 30001 erreichen. Der Installationsassistent wird Sie bitten, einen Namen und ein Kennwort zu vergeben, danach können Sie losbloggen.

Die einzelnen YAML-Abschnitte sind erklärungsbedürftig. Los geht es mit dem Namespace `backend`. Die WordPress-Installation in zwei Namespaces für Frontend und Backend zu teilen wäre in einem richtigen Cluster übertrieben – das soll in unserem Beispiel nur demonstrieren, wie Services zwischen Namespaces vermitteln. In der Backend-Definition folgt ein Deployment für die Datenbank mit nur einer

WordPress-Frontend

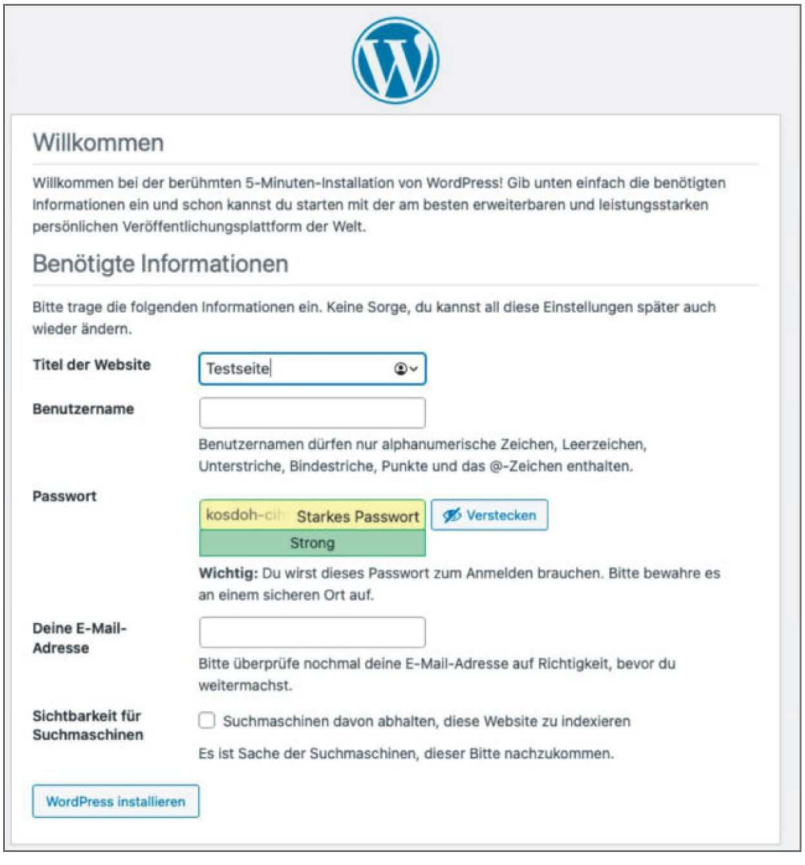
```yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: frontend
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wb-web-deployment
  namespace: frontend
  labels:
    app: wp-web
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 50%
      maxSurge: 1
  selector:
    matchLabels:
      app: wp-web
  template:
    metadata:
      labels:
        app: wp-web
    spec:
      containers:
        - name: web
          image: wordpress:latest
          ports:
            - containerPort: 80
          env:
            - name: WORDPRESS_DB_HOST
              value: "wp-database.backend"
            - name: WORDPRESS_DB_NAME
              value: "wp"
            - name: WORDPRESS_DB_USER
              value: "wp"
            - name: WORDPRESS_DB_PASSWORD
              value: "secretWp"
---
apiVersion: v1
kind: Service
metadata:
  name: wp-external
  namespace: frontend
spec:
  type: NodePort
  selector:
    app: wp-web
  ports:
    - protocol: TCP
      nodePort: 30001
      port: 80
      targetPort: 80
```

Kopie. Im Fall von MariaDB können Sie die Zeile nicht einfach zu `replicas: 3` ändern und sich an einer redundanten Datenbank erfreuen. Kubernetes würde drei Pods starten, aber MariaDB ist von Haus aus nicht darauf vorbereitet, im Team zu arbeiten.

Die Deployment-Strategie `Recreate` ist eine andere als das `RollingUpdate`, das Sie bereits kennengelernt haben. Mit der Einstellung `Recreate` stoppt Kubernetes bei einem Update den alten Container und erzeugt dann erst einen neuen. Das ist später entscheidend, wenn Sie der Datenbank persistenten Speicher zuweisen. Zwei gleichzeitig laufende MariaDB-Instanzen, die auf einen Ordner zugreifen, verursachen Datensalat! Sie merken es schon: MariaDB (wie auch MySQL) ist nicht gerade Cloud-native. Für redundanten Betrieb im Cluster brauchen Sie eine dafür geeignete Datenbank wie die Open-Source-Software CockroachDB (siehe ct.de/wwhr).

Das Deployment enthält einen Abschnitt, der Docker-Kennern sofort bekannt vorkommt: Der Abschnitt `env:` entspricht dem, was in einer Docker-Compose-Datei `environment:` heißt. Der Container bekommt Umgebungsvariablen. Um Missverständnissen vorzubeugen: Das Root-Kennwort der Datenbank im Klartext in die YAML-Datei zu schreiben ist nicht der letzte Schrei in der Kubernetes-Welt, für dieses Beispiel aber ausreichend.

Der nächste Abschnitt im Backend ist ein Service namens `wp-database`, der auf den Pod mit dem Label `app: wp-database` verweist. Für den Service ist kein Typ angegeben, Kubernetes weist ihm dann den Standard-Typ `ClusterIP` zu. Die Folge: Der Service ist



WordPress ist das ideale Beispiel für eine Anwendung, die aus mehreren Komponenten besteht: dem Webserver und einer Datenbank. Wenn die Einrichtungsseite erscheint, hat WordPress Zugriff auf seine Datenbank.

Cluster-intern erreichbar, wird aber nicht auf einen externen Port weitergereicht – so soll es bei einer Datenbank auch sein.

Die Definition des Frontends ist weitestgehend unspektakulär. Es gibt einen Namespace, ein Deployment für WordPress selbst (mit `RollingUpdate` und drei Kopien) sowie einen Service vom Typ `NodePort`, damit die Website auf Port 30001 der externen Netzwerkkarte veröffentlicht wird. Der Querverweis zwischen Frontend und Backend findet bei der Definition der Umgebungsvariable `WORDPRESS_DB_HOST` statt. Sie erhält den Wert `wp-database.backend`. Den Rest können Sie guten Gewissens dem Kubernetes-DNS-Server überlassen. Das WordPress-Frontend bekommt die IP-Adresse des Service aus dem anderen Namespace und kann mit seiner Datenbank kommunizieren.

Mit diesem Wissen nähern Sie sich in großen Schritten einem Kubernetes-Cluster, der mehr als nur Testseiten anzeigen kann und eine richtige Aufgabe erfüllt, indem er eine dynamisch erzeugte Website darstellt. Ein wesentlicher Makel besteht aber noch: Auf Port 30001 wird kein Besucher nach einer Website suchen. Und wünschenswert wäre später zusätzlich eine sichere TLS-Verbindung. Um diese Wünsche zu erfüllen, sollten Sie einen Reverse-Proxy einsetzen, der Anfragen von außen annimmt (später auch mit TLS) und an den passenden Container zustellt. Mit einem solchen Reverse-Proxy ist es auch möglich, verschiedene Dienste hinter einer öffentlichen IP-Adresse anzubieten. Die Software wertet den Anfrage-Header aus und kann zum Beispiel Anfragen an www.example.org an einen anderen Dienst im Cluster leiten als Anfragen an www.example2.org.

Ein solcher Reverse-Proxy, der sich im Cloud-Native-Umfeld einiger Beliebtheit erfreut und unter Open-Source-Lizenz veröffentlicht ist, heißt Traefik.

Um ihn im Cluster zu platzieren, könnten Sie eine sehr lange YAML-Datei aus der Traefik-Dokumentation herunterladen und per Apply-Befehl anwenden. Doch es gibt einen besseren Weg im Kubernetes-Universum.

Schöner mit Helm

Das kleine Kommandozeilenwerkzeug Helm ist angetreten, um die Installation von Software im Kubernetes-Cluster zu vereinfachen, selbst bezeichnet sich Helm als Paketmanager für Kubernetes. Die Grundidee: Die YAML-Definitionen für eine Anwendung werden mit Platzhaltern versehen (Helm arbeitet mit einer Template-Engine), zu einem Paket verschnürt (das in der Helm-Welt als Chart bezeichnet wird) und in einem Repository (technisch ist das nur ein Webserver) veröffentlicht. Der Nutzer installiert Helm auf seiner Maschine und installiert dann per Kommandozeile Anwendungen aus dem Repository in seinem Cluster. So viel der Theorie – Zeit, Traefik per Helm zu installieren.

Los geht die Helm-Karriere mit der Installation auf der lokalen Entwicklermaschine. Auf dem Mac mit `brew install helm`, unter Ubuntu mit `snap install helm --classic` und auf einer Windows-Maschine, sofern installiert, per Chocolatey: `choco install kubernetes-helm`. Dass das Ubuntu-Snap-Paket aktuell ein Jahr hinter der neuesten Helm-Version herhinkt, ist verschmerzbar. Wer keinen dieser Paketmanager nutzt, lädt die Binärdatei für alle Betriebssysteme von der Seite helm.sh herunter und legt sie per Hand in den Pfad für Programme – auch bei der Installation per Snap unter Ubuntu war das bei unserem Test nötig. Auf der Kommandozeile starten Sie Helm nach der Installation mit dem Befehl `helm`.

Mit Zugangsdaten für Ihren Cluster muss Helm nicht ausgestattet werden. Die Entwickler haben es sich und Ihnen einfach gemacht und nutzen einfach die Konfigurationsdatei und auch die Kontexte von Kubectl. Wenn Sie mehrere Cluster betreuen, wechseln Sie also per Kubectl zwischen diesen und arbeiten dann mit Helm.

Der erste Helm-Befehl, den man kennen muss, kommt Linux-Nutzern bekannt vor. Er zeigt, welche Anwendungen schon per Helm in einem Cluster installiert wurden:

```
helm ls
```

Wenn der Befehl eine leere Liste und keinen Fehler ausgibt, klappt die Verbindung zum Cluster und Sie können Traefik installieren. Die Traefik-Entwickler betreiben einen eigenen Server für Ihre Helm-Pakete, den man Helm als Repository bekannt machen muss:

```
helm repo add traefik ↵
      ↳https://helm.traefik.io/traefik
```

Es ist empfehlenswert, die Liste verfügbarer Charts vorab zu aktualisieren:

```
helm repo update
```

Dann kann Traefik den Weg in den Cluster finden:

```
helm install traefik traefik/traefik
```

Der Unterbefehl `helm install` erwartet zwei Informationen – zunächst einen Namen, den man frei vergeben kann (in diesem Fall schlicht `traefik`). Die Angabe `traefik/traefik` weist Helm an, das Paket traefik aus dem gleichnamigen Repository zu installieren. Ist der Befehl abgesetzt, zeigt `helm ls` einen Eintrag an und verrät, welche Version installiert wurde. Um Software per Helm zu aktualisieren, gibt es den Befehl `helm upgrade`, der wie `helm install` funktioniert:

```
helm upgrade traefik traefik/traefik
```

Den Erfolg der Helm-Installation können Sie mit vertrauten Kubectl-Mitteln sichtbar machen. Der Be-

```
❯ helm upgrade traefik traefik/traefik
Release "traefik" has been upgraded. Happy Helming!
NAME: traefik
LAST DEPLOYED: Thu Nov  3 19:09:29 2022
NAMESPACE: default
STATUS: deployed
REVISION: 2
TEST SUITE: None
NOTES:
Traefik Proxy 2.9.4 has been deployed successfully
on default namespace !
```

Helm bezeichnet sich selbst als Paketmanager. Mit dem Kommandozeilenprogramm installieren und aktualisieren Sie recht bequem Kubernetes-Pakete. Eine Software wie Traefik ist damit schnell installiert.

```
---
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: wordpress-ingress
  namespace: frontend
spec:
  entryPoints:
    - web
  routes:
    - match: PathPrefix(`/`)
      kind: Rule
      services:
        - name: wp-external
          port: 80
```

Traefik nutzt eine CRD vom Typ IngressRoute, um Routen für eingehenden Verkehr zu speichern. Darüber steuern Sie, welche Anfragen an welchen Service weitergeleitet werden.

fehl `kubectl get pods` zeigt zum Beispiel, dass das Traefik-Paket einen Pod angelegt hat. Außerdem gibt es einen Service, der schon mal die Ports 80 und 443 abhört. Dass Traefik schon arbeitet, sehen Sie, wenn Sie eine IP-Adresse Ihres Clusters im Browser öffnen. Die Fehlermeldung „404 page not found“ kommt bereits von Traefik, weil noch keine Routen eingerichtet sind.

Wege hinein

Im letzten Schritt in diesem Teil der Reihe veröffentlichen Sie die WordPress-Installation hinter Traefik. Dafür sind zunächst ein paar Änderungen am Service `wp-external` im Namespace `frontend` nötig. Streichen Sie in der YAML-Definition die Zeile `type: NodePort` und machen ihn damit zu einem internen Service vom Typ `ClusterIP`. Auch die Zeile `nodePort: 30001` muss verschwinden. Traefik allein muss künftig auf interne Services zugreifen. Speichern Sie die Änderungen an der Datei und wenden sie mit Kubectl an.

Damit Traefik weiß, was es mit einer eingehenden Anfrage anstellen soll, brauchen Sie ein neues Kubernetes-Objekt, das Sie wie gewohnt an eine bestehende Datei anhängen oder in einer neuen Datei definieren können. Das Objekt bekommt den Typ `IngressRoute` – das ist kein Objekt, das zum Repertoire von Kubernetes gehört. Traefik greift hier zu einem ausgesprochen mächtigen Kubernetes-Konzept: der Custom Resource Definition (CRD). Die CRD wurde angelegt, als Sie Traefik via Helm installiert haben. Anwendungen können das Kubernetes-API per CRDs erweitern, um eigene Konfigurationen darin abzulegen. Die IngressRoute für den WordPress-Service finden Sie unten sowie über ct.de/wwhr zum Download. Sie enthält lediglich eine sehr einfache Regel: Sämtliche Anfragen sollen beim Service `wp-external` ankommen. Sofern Sie öffentliche DNS-Einträge für eine Domain, die Sie kontrollieren, eingerichtet haben, die auf die externen IP-Adressen Ihres Clusters zeigen, können Sie auch nach angefragter Domain filtern:

```
- match: Host(`www.example.org`) ||↵
                ↳ Host(`example.org`)
```

Doch Traefik kann noch mehr – und Anfragen zum Beispiel autorisieren. Mehr dazu finden Sie in der Dokumentation der Software (siehe ct.de/wwhr).

Resümee

Ihr Cluster nimmt langsam Form an und liefert eine dynamische Website auf Port 80 aus. Mit Helm kennen Sie außerdem den Schlüssel, um Software anderer Entwickler im Cluster einzusetzen und aktuell zu halten – ein weiterer Baustein auf dem Weg zu einer brauchbaren Produktivumgebung. Ein paar Baustellen sind aber noch offen: Zunächst ist Ihre WordPress-Instanz noch nicht sonderlich langlebig. Immer, wenn der Datenbank-Container ersetzt wird, sind auch alle Daten weg. Es fehlt an persistentem Speicher. Im nachfolgenden vierten Teil dieser Reihe wird dieses Problem gelöst – denn wie bei fast allen Aufgaben hat das Kubernetes-Ökosystem dafür ein paar sehr umfangreiche Lösungen zu bieten. (jam) c't

YAML-Dateien und Dokumentation

ct.de/wwhr

» Continuous Lifecycle »

[Container Conf]

15. – 16. November 2023 in Mannheim

Software effizienter entwickeln und betreiben

Die **Continuous Lifecycle/ContainerConf** liefert vertiefende Einblicke in relevante Fachthemen entlang des gesamten Softwarelebenszyklus – mit einem Fokus auf **Developer Experience** und optimal integrierte **Platform Operations.**

Workshops und Vorträge widmen sich Prozessen und Tools von der Containerisierung über **Continuous Delivery** und **DevOps** – inklusive **DevSecOps** und **GitOps** – bis hin zum professionellen Betrieb der Anwendungen.

Jetzt Tickets zum **Frühbucher-Rabatt** sichern!

Die Konferenz richtet sich an:

- Infrastruktur-Engineers und -Architekt:innen
- Softwareentwickler:innen
- Softwarearchitekten:innen
- Team- und Projektleiter:innen
- IT-Berater:innen

Jetzt anmelden: www.continuouslifecycle.de

Veranstalter

dpunkt.verlag

heise Developer

Gold-Sponsor

T Systems

Silber-Sponsoren

andrena OBJECTS

Giant Swarm

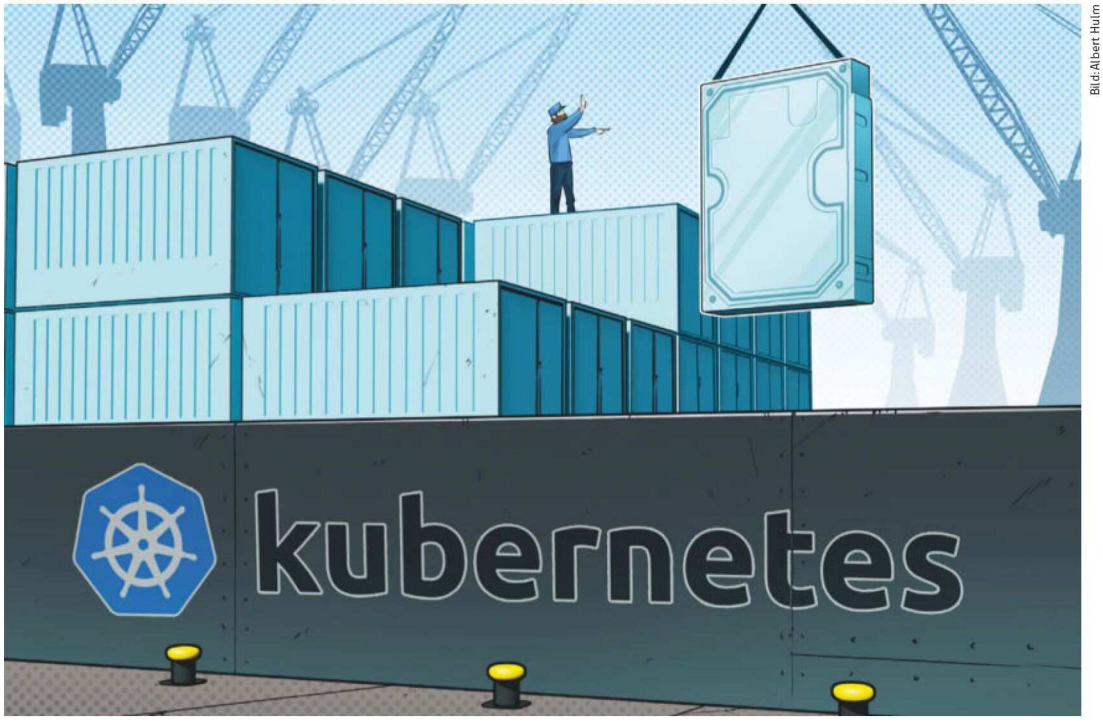


Bild: Albert Hulm

Volumes, Secrets und ConfigMaps

Container sind flüchtige und vergängliche Gebilde, erzeugt aus Abbildern. Damit sie Daten dauerhaft speichern können, brauchen sie Volumes für Dateien und Ordner. Was in der Docker-Welt mit einem Einzeiler abgehakt ist, ist im Kubernetes-Cluster kompliziert – dafür aber perfekt steuerbar.

Von **Jan Mahn**

Einfach ist alle Containerei, solange die Software, die im Container steckt, nichts speichern will. Dann ist es egal, wo man die Container startet und ob man sie wegwirft, ersetzt oder klont. „Stateless“ heißen solche Anwendungen in der Werbesprache der Cloudanbieter. Der Haken: Nur die wenigsten Anwendungen sind wirklich stateless, fast immer gibt es zur Laufzeit etwas zu speichern oder von der Festplatte zu lesen.

Weil das so ist, haben die Entwickler des Container-Orchestrators Kubernetes für solche Fälle vorgesorgt und sehen Schnittstellen vor, über die Con-

tainer an Speicherplatz kommen. In einem Cluster aus mehreren Servern kann das schnell kompliziert werden – daher mussten Sie bis zu diesem vierten Teil unserer Reihe für Kubernetes-Einsteiger warten, bis Ihre Pods persistenten Speicher bekommen.

In der Docker-Welt ist der Umgang mit Speicherplatz eine einfache Aufgabe. Docker unterscheidet zwischen zwei Arten von Volumes, benannten und unbenannten. Bei benannten Volumes kümmert sich der Docker-Daemon um einen zentralen Speicherort und führt für jedes benannte Volume ein Objekt in seiner internen Datenhaltung. Darauf kann man mit Docker-Kommandozeilenbefehlen wie `docker volume ls` zugreifen. Unbenannte Volumes dagegen sind eine direkte Verknüpfung zwischen einer Datei oder einem Ordner auf der lokalen Platte zu einem Pfad in einem Container („Bind Mounts"). Eine gängige Aufgabe für unbenannte Volumes sind in der Docker-Welt alle Formen von Konfigurationsdateien. In einer Docker-Compose-Datei sieht das zum Beispiel so aus:

```
services:
  web:
    image: nginx:alpine
    ports:
     - 80:80
    volumes:
     - ./config/nginx.conf:↵
                ↳/etc/nginx/nginx.conf
```

Docker-Compose arbeitet immer relativ zum Speicherort der Datei docker-compose.yml, in diesem Beispiel muss im selben Ordner also der Ordner config mit der Konfigurationsdatei nginx.conf liegen. Dann hat der Container Zugriff auf diese Datei. Fehlt die Datei, legt Docker-Compose ein leeres Verzeichnis dieses Namens an, um die Anforderung zu erfüllen.

Mit Kubernetes funktioniert ein solches Konzept nicht, weil die Maschine, auf der Sie Ihre YAML-Dateien zum Verwalten des Clusters schreiben, völlig unabhängig vom Kubernetes-Cluster arbeitet und der Cluster nach einem Kubectl-Apply-Befehl keinerlei Kontakt mehr mit dem PC des Administrators hat.

Konfigurationsdateien

Die Kubernetes-Alternative für solche Aufgaben ist die ConfigMap. Das ist ein eigenständiges Kubernetes-Objekt (wie ein Pod oder ein Service), das erst in den Cluster gebracht wird, dort wie alle Objekte in der Kubernetes-Datenbank gespeichert wird und dann in einen oder mehrere Pods eingebunden wird. Von der lokalen Maschine, auf der die zugehörige Datei entstand, ist das gänzlich entkoppelt. Eine vereinfachte ConfigMap für eine Nginx-Konfiguration kann zum Beispiel so aussehen:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: configmap-nginx-example
data:
  nginx.conf: |
    user www www;
    worker_processes 5;
    http {
    [...]
    }
```

Zum Einsatz kommt dabei ein Trick, den YAML von Haus aus mitbringt. Der Block-Operator | sagt dem YAML-Parser: „Behandle die folgenden (eingerückten) Zeilen als eine Zeichenkette und erhalte die Zeilenumbrüche." In der ConfigMap liegt ein Schlüssel namens `nginx.conf` mit dem Inhalt der Konfigurationsdatei als Wert. Die Inhalte sind kein YAML, sondern in diesem Fall die Nginx-eigene Syntax für diese Datei. Es wäre aber nicht verboten (und durchaus gängig), auf diese Weise YAML in YAML zu verschachteln, wenn eine Anwendung YAML verlangt.

Nicht immer muss der Inhalt einer ConfigMap ein solcher mit | eingeleiteter Block sein, man kann auch einen einfachen Wert in einer ConfigMap speichern. Und auch mehrere Schlüssel sind unterhalb von `data:` möglich. Auch das Folgende ist eine gültige ConfigMap:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-example
data:
  timeout: 5
  username: demo-user
```

Sobald eine ConfigMap im Cluster liegt, kann man sie an Pods anhängen und in den Containern im Pod nutzen. Auch dafür gibt es mehrere Varianten. Die gängigste: Der Inhalt eines Schlüssels in der ConfigMap soll als Datei im Dateisystem des Containers landen – im Beispiel geht es um den Schlüssel `nginx.conf`, der in einem Nginx-Container als

gleichnamige Datei erwartet wird. Die passende YAML-Datei für den Pod sieht so aus:

```
kind: Pod
apiVersion: v1
metadata:
  name: nginx-with-config

spec:
  volumes:
    - name: nginx-config
      configMap:
        name: configmap-nginx-example
  containers:
    - name: webserver
      image: nginx:alpine
      volumeMounts:
        - name: nginx-config
            mountPath: /etc/nginx
```

Der überwiegende Teil ist vertraute Materie. Neu ist die zusätzliche Angabe `volumes:` unterhalb von `spec:`. Auf der Ebene des Pods (damit also für alle Container gültig) können Volumes deklariert werden. Konkret entsteht im Beispiel ein Volume namens `nginx-config`, mit den Inhalten aus der ConfigMap namens `configmap-nginx-example`, die zuvor angelegt wurde. Mit dem Deklarieren des Volumes im Pod allein ist es aber noch nicht getan, es muss anschließend an einen oder mehrere Container in diesem Pod geheftet werden. Das passiert unterhalb von `volumeMounts:` auf der Ebene des Nginx-Containers. Der Name `nginx-config` ist der, der zuvor innerhalb des Pods definiert wurde, der `mountPath` ist der Zielordner im Container. Kubernetes erzeugt jetzt für jeden Schlüssel in der ConfigMap eine Datei und legt dort die Inhalte hinein – der Nginx-Container erhält somit seine Datei nginx.conf.

Mit einem solchen Konstrukt haben Sie es im Kubernetes-Alltag recht häufig zu tun, weil viele Anwendungen über Konfigurationsdateien verwaltet werden.

Variable Umgebung

Der zweite gängige Weg im klassischen Linux-Umfeld: Umgebungsvariablen. Auch die kann man – muss man aber nicht – über ConfigMaps befüllen. In dem Fall muss man die ConfigMap nicht auf Pod-Ebene einbinden. Stattdessen verweist man direkt in der Container-Beschreibung auf Schlüssel aus einer ConfigMap. Der folgende Schnipsel veranschaulicht das anhand eines MariaDB-Containers, der eine Umgebungsvariable aus einer ConfigMap bezieht:

```
[...]
spec:
  containers:
   - name: mariadb
     image: mariadb
     ports:
      - containerPort: 3306
     env:
      - name: MARIADB_USER
        valueFrom:
        configMapKeyRef:
          name: mariadb-config
          key: MARIADB_USER
```

Der Container bekommt die Umgebungsvariable `MARIADB_USER` aus dem Schlüssel `MARIADB_USER`, der in der ConfigMap `mariadb-config` liegen soll. Dieser Schreibweise begegnet man in YAML-Dateien von großen Projekten hin und wieder, häufiger sieht man jedoch in solchen Fällen die Angabe `envFrom:`. Damit spart man es sich, einzelne Schlüssel aus einer ConfigMap den jeweiligen Umgebungsvariablen zuzuweisen. Kubernetes nimmt dann einfach alle Werte, die unterhalb von `data:` in der ConfigMap liegen und pflanzt sie als Umgebungsvariablen ein. Das sieht zum Beispiel so aus:

```
[...]
spec:
  containers:
   - name: mariadb
     image: mariadb
     envFrom:
      - configMapRef:
        name: mariadb-config
```

Um die Konstruktion in Aktion zu sehen, werfen Sie einen Blick auf die YAML-Beispiele zu diesem Artikel, die wir über ct.de/wyhx bereitstellen. Das WordPress-Beispiel aus Teil 3 dieser Reihe wird dort per ConfigMap konfiguriert.

Geheimnisse im Cluster

Wenn Sie das Prinzip von ConfigMaps verinnerlicht haben, können Sie ein damit verwandtes Objekt ebenfalls einsetzen: Für sensible Inhalte wie Kennwörter, Token und Zertifikate nutzt man statt einer

ConfigMap das Secret. Angelegt und eingesetzt wird ein Secret fast identisch. Der wesentliche Unterschied: Die Inhalte unterhalb von `data:` müssen als Base64-Zeichenkette enkodiert sein. Ein Secret, das den Benutzernamen `wp` und das Kennwort `sicher` für eine Datenbank enthält, legt man zum Beispiel so an:

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-wp-db
type: Opaque
data:
  username: d3A=
  password: c2ljaGVy
```

Eine Zeichenkette ist in unixoiden Betriebssystemen schnell auf der Kommandozeile in Base64 enkodiert:

```
echo -n 'sicher' | base64
```

Weil es oft verwechselt wird, noch einmal der Hinweis: Base64 ist ein Kodierungsverfahren, keine Verschlüsselung oder Hash! Es geht bei der Umwandlung also nicht darum, die Sicherheit zu erhöhen. Base64 stellt lediglich sicher, dass eine Zeichenkette mit Sonderzeichen, Zeilenumbrüchen und ähnlichen Zeichen fehlerfrei übertragen wird.

Wer ein Secret ohne vorherige Base64-Umwandlung definieren will, schreibt statt `data:` einfach `stringData:` und die Inhalte als lesbare Zeichenkette. Setzt man beide Schlüssel untereinander, verwendet Kubernetes die Inhalte von `stringData:`.

Ein anderes Missverständnis: Von Haus aus ist ein Secret nicht sicherer als eine ConfigMap. Solange es im Cluster keine Berechtigungsverwaltung und nur einen Nutzer mit Vollzugriff gibt, kann dieser den Inhalt auch wieder sichtbar machen. Dafür reichen die Kubectl-Bordmittel:

```
kubectl get secret secret-wp-db↵
                    ↳ --output=yaml
```

Eine Base64-Zeichenkette ist schnell zurückübersetzt:

```
echo "c2ljaGVy" | base64 --decode
```

Konfigurationen und Geheimnisse in Secrets und ConfigMaps zu trennen, zahlt sich dennoch später aus. Wenn Sie sich irgendwann mit Berechtigungsverwaltung im Cluster auseinandersetzen, können Sie den Zugriff auf Secrets einfach für bestimmte Nutzer verhindern. Eher ein Nischenproblem löst dagegen die Funktion „encryption at rest“. Kubernetes und auch k3s kennen eine Möglichkeit, die Secrets verschlüsselt in der darunterliegenden etcd-

```
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-wp-db
  namespace: backend
spec:
  storageClassName: local-path
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 4Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wb-database-deployment
  namespace: backend
  labels:
    app: wp-database
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: wp-database
  template:
    metadata:
      name: wp-database
namespace: backend
```

Datenbank abzuspeichern (siehe ct.de/wyhx). Dann kann man den Klartext auch dann nicht extrahieren, wenn man sich des Servers bemächtigt hat und an Kubernetes vorbei aufs Dateisystem zugreifen kann. In normalen Umgebungen (in der nur wenige Administratoren überhaupt auf den Server zugreifen), ist das nicht nötig.

In einen Container kommen die Inhalte von Secrets und ConfigMaps per Umgebungsvariable oder als Volume. Der folgende Beispiel-Schnipsel zeigt gleich beide Varianten:

```
[...]
spec:
  volumes:
    - name: secret-volume
      secret:
        secretName: secret-example
  containers:
   - name: mariadb
     image: mariadb
     env:
      - name: MARIADB_PASSWORD
        valueFrom:
          secretKeyRef:
            name: secret-wp-db
            key: password
     volumeMounts:
      - name: secret-volume
        mountPath: /etc/example
```

Das Passwort wird als Umgebungsvariable injiziert und im Ordner /etc/example liegen Dateien, die in einem Secret namens `secret-example` definiert wurden. Im Container müssen Sie sich in beiden Fällen nicht mehr mit Base64 herumschlagen, Kubernetes übersetzt automatisch wieder zurück.

Wenn Sie den Einsatz von Secrets und ConfigMaps am praktischen Beispiel nachvollziehen wollen, finden Sie eine umgebaute Version des WordPress-Beispiels aus Teil 3 dieser Reihe über ct.de/wyhx. Konfigurationen liegen in ConfigMaps, Geheimnisse in Secrets. Die Deployments heißen wie vorher, Kubernetes wird bestehende Objekte also ersetzen und keine zweite Instanz starten.

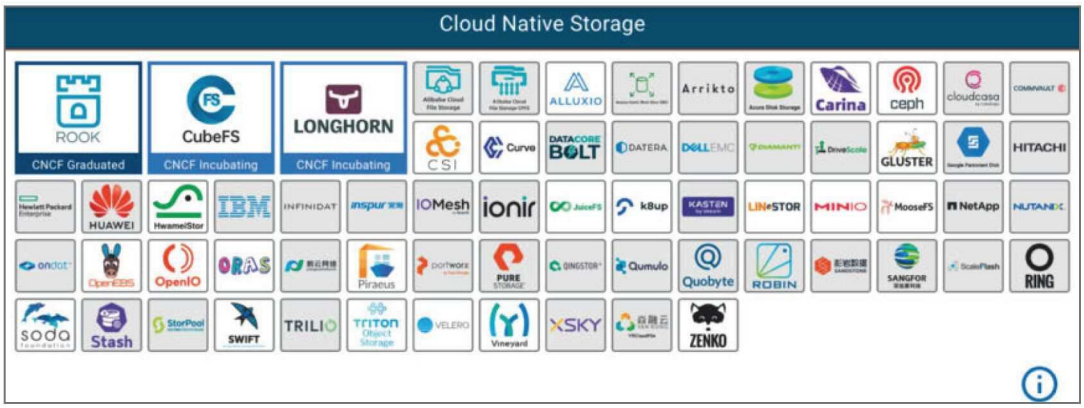
Echte Volumes

Genug der schnöden Konfigurationsdateien und Umgebungsvariablen – in diesen Artikel gelockt haben wir Sie mit dem Versprechen, auch solche Daten im Cluster zu speichern, die im Betrieb einer Anwendung anfallen. Eine Datenbank wie MariaDB ist dafür ein gutes Beispiel. Ohne persistenten Speicher ist sie ziemlich nutzlos, weil sie bei jedem Update oder Neustart mit einem leeren Verzeichnis beginnt und zunächst eine leere Datenbank anlegt.

Der Zugriff auf das Dateisystem ist in Kubernetes gleich hinter zwei Abstraktionsschichten versteckt. Doch kein Grund zur Panik, mit deren Konfiguration hat man am Anfang nur wenig zu tun. Auf der untersten Ebene kennt Kubernetes das Konzept der StorageClass. Dabei handelt es sich um ein Kubernetes-Objekt, das Konfigurationsdaten für eine Klasse von Speicherplatz definiert. Die zentrale Information in diesem Objekt ist der Name eines Provisioners. Dabei handelt es sich um einen Verweis auf eine containerisierte Software, die wie ein Treiber oder Adapter funktioniert und mit einem Speichergerät im Hintergrund kommuniziert. Der Speicherplatz selbst kann dabei im Cluster liegen, irgendwo im lokalen Netz oder auch im Internet. Der Provisioner vermittelt zwischen verschiedenen Techniken und kümmert sich darum, dass am anderen Ende des Adapters immer ein neutrales Dateisystem ankommt. Wie ein Provisioner im Detail funktioniert, müssen Sie als Anwender nicht bis ins Detail durchdringen. Ihre Aufgabe besteht darin, einen passenden Provisioner zu wählen.

Wer in einem Unternehmensnetz zum Beispiel ein bestehendes NFS hat und seine Nutzdaten dort, also außerhalb des Clusters ablegen will, greift zu einem NFS-Provisioner. Provisioner gibt es auch für andere in Unternehmen verbreitete Systeme wie vSphere oder Ceph (siehe ct.de/wyhx). Wer dagegen seinen Cluster bei einem der drei großen Cloudprovider (Amazon AWS, Google Cloud oder Microsoft Azure) betreibt, bekommt von seinem Provider einen Provisioner, der mit den hauseigenen Block-Storage-Systemen kommuniziert. Wie man solche Provisioner einsetzt, würde diesen Artikel sprengen – die zugehörige Dokumentation finden Sie über ct.de/wyhx.

In einem selbst betriebenen Cluster, den Sie, wie im ersten Teil der Reihe beschrieben, mit der Kubernetes-Distribution k3s angelegt haben, ist ein Provisioner bereits angelegt und auch eine zugehörige StorageClass steht direkt bereit. Der Provisioner namens `local-path` funktioniert denkbar einfach: Er schnappt sich den Ordner /var/lib/rancher/k3s/storage auf den Nodes und legt dort Volumes an, die Sie an Container hängen können. Das entspricht ziemlich genau dem, was Sie von einem benannten Volume aus der Docker-Welt bekommen.



Die CNCF-Landkarte (landscape.cncf.io) zeigt, wie vielfältig das Angebot an Storage-Anbindungen für Kubernetes ist. Per StorageClass und PersistentVolumeClaim verbindet man sie mit Containern.

Zwischen StorageClass und Pod haben die Entwickler noch eine Abstraktionsschicht gestellt – den PersistentVolumeClaim (PVC). Mit einem solchen Objekt bestellt man zunächst ein Volume beim Provisioner und verweist im Pod dann auf den Namen dieses PVC. Klingt theoretisch kompliziert, wird am praktischen Beispiel aber schnell klar. Im Kasten „Der Kubernetes-Lernpfad“ auf Seite 66 sehen Sie, wie ein Datenbank-Container seinen Speicherplatz vom Provisioner `local-path` bekommt.

Es ist nicht immer ratsam, zu den Ersten zu gehören, die eine neue Version ausprobieren.

Das erste Objekt beschreibt einen PVC mit dem Namen `pvc-wp-db`. Bestellt werden 4 GByte Speicherplatz. An dieser Angabe können Sie direkt einen Vorteil der Abstraktionsschicht erkennen: Weil der maximale Platz vorab bestellt wird, kann der Provisioner direkt auf die Bestellung reagieren. Wenn seine Festplatten im Hintergrund zum Beispiel voll sind und er den Wunsch nicht erfüllen kann, wird er den PVC ablehnen – dann wird auch der Container nicht erstellt und Sie bekommen direkt eine Fehlermeldung. Das ist besser als ein vollgelaufenes Laufwerk, das Sie als Admin in einigen Monaten ausgerechnet am Freitagabend überrascht. Andere Möglichkeiten von PVCs für Fortgeschrittene: Wenn Sie in einem Unternehmen einen großen Cluster verwalten, können Sie zum Beispiel

Regeln erlassen, welches Team wie viel Speicherplatz konsumieren darf (was ja in der Regel mit Kosten verbunden ist).

In Ihrem eigenen Cluster gibt es solche Restriktionen nicht, der PVC ist erfüllbar und wird angelegt. Zum Einsatz kommt er im Pod für die Datenbank. Auf Ebene des Pods (also für alle Container) wird der PVC als Volume eingebunden und als `volume-wp-db` Pod-intern bekannt gemacht. Das allein reicht aber immer noch nicht, damit die Daten auch im Container ankommen. Diese Zuordnung von Volume zu Pfad geschieht zu guter Letzt auf Container-Ebene unterhalb von `volumeMounts:`, wie Sie es von Secrets und ConfigMaps bereits kennen.

Damit sind Sie bereit, eine WordPress-Instanz mit einer persistenten Datenbank zu betreiben. Wenn Sie das Beispiel nachvollziehen wollen, laden Sie die YAML-Definition über ct.de/wyhx herunter und bringen Sie es in den Cluster. Wenn dort noch eine WordPress-Instanz aus Teil drei dieser Reihe läuft, wird sie aktualisiert. Wenn nicht, wird sie neu angelegt.

Den Erfolg der Umstellung können Sie leicht überprüfen: Löschen Sie den Datenbank-Pod mit `kubectl delete pod <Name>` und warten Sie darauf, dass aus dem Deployment ein neuer Pod entsteht. Die Daten überleben jetzt im Volume.

Redundanter Speicherplatz

Der Local-Path-Provisioner ist einfach zu handhaben, doch er hat auch Schwächen. In einem Cluster aus mehreren Nodes wird der Provisioner die Daten nur auf einem Server lokal ablegen und sich dann darum kümmern, dass Pods, die das Volume brauchen, immer auf dieser Maschine platziert werden. Fällt ausgerechnet die Maschine mit der Datenbank aus, kann Kubernetes den Pod nicht woanders starten und die Anwendung fällt aus.

Besser wird die Welt erst mit einem anderen Provisioner, der in der Lage ist, Daten über mehrere Server redundant zu halten. Das ist keine ganz triviale Aufgabe, weil auf dem Weg viel schiefgehen kann (Konflikte, Paketverluste, Ausfälle). Ein Provisioner, der mit solchen Widrigkeiten umgehen kann, heißt Longhorn, ist Open-Source-Software und stammt aus dem Hause Rancher (die Firma, die ursprünglich auch mal k3s erfunden und dann an die CNCF übergeben hat). Und Longhorn kann noch mehr als redundante Datenhaltung, eingebaut ist auch ein Backup-Mechanismus, der die Inhalte von Volumes auf externe Datenhalden (zum Beispiel S3-Speicher) kopieren kann. Wie Sie Longhorn in Betrieb nehmen, lesen Sie im Artikel „Redundanter Speicher mit Longhorn“ ab Seite 76.

Raus aus dem Dickicht

Nach dieser Einführung in Volumes und Konfigurationsdateien können Sie langsam das Dickicht der Kubernetes-Objekte verlassen und sind bereit, sich auf der weiten Ebene nützlicher Cloud-Native-Helfer umzusehen. Den wichtigsten Kubernetes-Ideen sind Sie jetzt begegnet, ab jetzt geht es vor allem darum, fertige Cloud-Native-Software zu konfigurieren und Routine im Umgang mit Deployment, PVC & Co. zu sammeln. Im Kasten „Der Kubernetes-Lernpfad“ oben sehen Sie, welche Schritte Sie bereits hinter sich gebracht haben und was es noch zu erkunden gibt. Im nachfolgenden fünften Teil dieser Reihe lernen Sie, wie Sie Ihren Cluster absichern und TLS einrichten – für die Arbeit mit Zertifikaten sind Volumes eine wichtige Voraussetzung, die Sie jetzt in der Tasche haben. (jam) c't

Der Kubernetes-Lernpfad

Wenn Sie dieser Kubernetes-Reihe gefolgt sind, haben Sie wesentliche Konzepte bereits kennengelernt. Ein Umzug von Docker-Anwendungen in einen Kubernetes-Cluster rückt langsam in greifbare Nähe. Gleichsam gibt es im Kubernetes-Umfeld noch einiges zu entdecken. Hier befinden Sie sich auf Ihrer Reise:

- ✓ Cluster, Node und Kubectl
- ✓ Pod, Container und Namespace
- ✓ Service und IngressRoute
- ✓ ConfigMap und Secret
- ✓ StorageClass und Persistent-VolumeClaim
- – TLS und Cluster-Sicherheit
- – eigene Anwendungen mit Helm verpacken
- – Logging und Monitoring
- – automatische Cluster-Verwaltung und GitOps
- – Anwendungen für Kubernetes entwickeln

Beispiele und Dokumentationen

ct.de/wyhx

WIR TEILEN KEIN HALBWISSEN
WIR SCHAFFEN FACHWISSEN

Einführung in den Kea DHCP Server

Der Workshop gibt eine vollständige Einführung in die neue Kea-DHCP-Software auf Unix- und Linux-Systemen. Sie lernen, wie man das Kea-DHCP-System installiert, konfiguriert und wartet.

Einführung in GitLab

Der Workshop bietet einen Einstieg in den Betrieb einer eigenen Gitlab-Instanz. Sie lernen GitLab initial aufzusetzen, sowie Ihre Instanz zu konfigurieren und an eigene Anforderungen anzupassen.

Docker und Container in der Praxis

Der Workshop für Entwickler und Administrierende behandelt neben theoretischem Wissen über Container auch Herausforderungen im Alltag und eigene Container-Erfahrungen auf der Kommandozeile.

CI/CD mit GitLab

Der zweitägige Workshop bietet eine praktische Einführung in die GitLab-CI-Tools und zeigt, wie man damit Softwareprojekte baut, testet und veröffentlicht.

Sichern Sie sich Ihren Frühbucher-Rabatt:

heise-academy.de/marken/ct

Bild: Albert Hulm

Sicherheit im Cluster

Wenn der erste Cluster läuft und die Kubectl-Befehle leicht von der Hand gehen, möchte man als Kubernetes-Einsteiger am liebsten mit den ersten produktiven Anwendungen beginnen. Doch vorher sollte man sich etwas Zeit nehmen für die Sicherheit von Cluster und Anwendern.

Von **Jan Mahn**

Geschafft – in den vorangegangenen Artikeln haben Sie eine Anwendung Schritt für Schritt in Kubernetes überführt. Bevor Sie das Wissen auf eigene Projekte anwenden, sollten Sie aber noch ein paar Sicherheitskonzepte kennenlernen und gleich von Anfang an einsetzen – das ist immer erfolgversprechender, als Security nachträglich an eine fertige Anwendung dranzubasteln. Auf den folgenden Seiten lernen Sie, wie Sie Ihren Cluster in drei Schritten sicherer machen: mit Transportverschlüsselung, Netzwerkregeln für Pods und Zugriffsberechtigungen für Admins.

Am Ende des vierten Teils dieser Reihe meldete sich eine WordPress-Instanz auf Port 80 per HTTP. Vor 15 Jahren hätte man damit noch Benutzer und Admins begeistern können, heute fehlt ein entschei-

dendes Detail: HTTPS mit einem gültigen Zertifikat, das von einer vertrauenswürdigen Stammzertifizierungsstelle stammt und dem die Browser vertrauen. Das muss man heute nicht mehr kaufen, Let's Encrypt liefert es kostenlos und der HTTP-Router Traefik, den Sie im Laufe der Reihe schon installiert haben, beschafft Zertifikate von diesem Anbieter und verlängert sie automatisch. Die erste Aufgabe, um diese Beschaffung einzurichten, hat nichts mit Kubernetes zu tun. Damit Sie ein Zertifikat bekommen können, müssen Sie einen A- und optional einen AAAA-DNS-Eintrag für eine Domain oder eine Subdomain setzen, der auf eine beliebige, externe IP-Adresse Ihres Clusters verweist. Erledigen Sie diese Aufgabe am besten mit etwas zeitlichem Abstand, damit sich der Eintrag in allen DNS-Caches herumspricht. Danach können Sie sich an die Vorbereitungen im Cluster machen.

Traefik mit Speicher

Wenn Sie den Beschreibungen dieser Kubernetes-Reihe gefolgt sind, haben Sie Traefik im dritten Teil mit folgenden Zeilen über den Kubernetes-Paketmanager Helm installiert:

```
helm repo add traefik ↵
↳https://helm.traefik.io/traefik
helm repo update
helm install traefik traefik/traefik
```

Helm legt ein Deployment mit dem Namen `traefik` aus dem Chart `traefik/traefik` an. Diese Installation müssen Sie für die automatische Zertifikatsbeschaffung etwas erweitern. Traefik braucht persistenten Speicherplatz für die Zertifikate und die Zertifikatsbeschaffung über Let's Encrypt braucht eine minimale Konfiguration. Immer, wenn es in einer Installation per Helm etwas einzurichten gibt, erledigt man das über eine Values-Datei. Das ist ein Stück YAML, das man Helm mit einem Install- oder Upgrade-Befehl unterschiebt. Wie das YAML aussieht, definieren die Entwickler des jeweiligen Helm-Pakets – man befüllt damit Variablen, die im Helm-Chart verwendet werden. Wie das im Detail funktioniert, müssen Sie erst durchdringen, wenn Sie eigene Anwendungen in Helm-Charts verpacken wollen. Als Anwender eines Charts reicht es, die benötigten Konfigurationen aus der Dokumentation in eine neue Datei zu legen und anzupassen. Für Traefik mit TLS legen Sie auf Ihrer lokalen Maschine eine Datei traefik-values.yaml an (den Namen können Sie frei vergeben). In die Datei kommen folgende Zeilen:

```
persistence:
  enabled: true
  name: data
  accessMode: ReadWriteOnce
  size: 128Mi
  path: /data
  annotations: {}

certResolvers:
  letsencrypt:
    email: <Ihre Mailadresse>
    tlsChallenge: true
    httpChallenge:
      entryPoint: "web"
    storage: /data/acme.json

ports:
  websecure:
    tls:
      certResolver: "letsencrypt"
  web:
    redirectTo: websecure
```

Wie auch in den ersten Teilen dieser Reihe finden Sie alle Inhalte zum Download in einem GitHub-Repository (siehe ct.de/w6xs), abtippen müssen Sie also nichts. Der erste Abschnitt `persistence:` aktiviert persistenten Speicherplatz. In der Folge legt Helm einen PersistentVolumeClaim an und bindet ihn an den Traefik-Container. 128 MByte reichen für die Zertifikate und Metadaten, die Traefik anlegt, absolut aus.

Der zweite Abschnitt konfiguriert die ACME-Funktion von Traefik. Über diesen Standard bezieht die Software das Zertifikat bei Let's Encrypt und legt dafür auf Port 80 (`entryPoint: "web"`) eine HTTP-Challenge ab, mit der Let's Encrypt prüft, ob Sie diese Domain kontrollieren. Ändern müssen Sie in diesem Beispiel nur Ihre Mailadresse. Sie wird nicht im Zertifikat veröffentlicht, Let's Encrypt nutzt sie nur, um Sie zu warnen, wenn das Zertifikat ausläuft – das sollte aber nicht passieren, wenn Ihr Server läuft: Traefik beschafft kommentarlos neue Zertifikate.

Der dritte Abschnitt `ports:` aktiviert schließlich den zuvor definierten Zertifikatsdienst namens `letsencrypt` für den HTTPS-Port, den Traefik `websecure` nennt. Der Endpunkt `web` wird mit `redirectTo` angewiesen, sämtlichen Verkehr auf seinen HTTPS-Kollegen umzuleiten. Ihre Nutzer surfen also immer mit TLS.

Liegt die Datei traefik-values.yaml auf Ihrer lokalen Maschine mit installiertem Helm bereit, navigieren Sie auf der Kommandozeile in den Ordner mit dieser Datei und aktualisieren Sie das Helm-Deployment mit diesen Werten:

```
helm upgrade -f traefik-values.yaml ↵
↳traefik traefik/traefik
```

Wenig später sollten Sie mit `kubectl get pvc` einen PersistentVolumeClaim für Traefik sehen. Damit sind die Voraussetzungen geschaffen, um eine Ingress-Route auf HTTPS umzustellen. Als Beispiel dient die Wordpress-Route aus Teil 3 und 4 der Reihe. Nur zwei Änderungen an der YAML-Datei sind nötig:

```
apiVersion: traefik.containo.us/↵
                              ↳v1alpha1
kind: IngressRoute
metadata:
  name: wordpress-ingress
  namespace: frontend
spec:
  entryPoints:
    - websecure # vorher: web
  routes:
      # vorher: match: PathPrefix(`/`)
    - match: Host(`www.example.org`)
      kind: Rule
      services:
        - name: wp-external
          port: 80
```

Die erste Änderung betrifft den `entryPoint`. Statt auf Port 80 soll die Seite künftig auf Port 443 antworten. Damit der Zertifikatsdienst von Traefik weiß, für welche Domains er ein Zertifikat ordern soll, brauchen Sie immer eine Regel vom Typ `Host` mit dem Namen. Das muss nicht nur eine sein: Mit dem Operator || können Sie auch mehrere Host-Einträge angeben (zum Beispiel www.example.org und example.org).

Um Ihre WordPress-Instanz umzustellen, ändern Sie die Zeilen in der YAML-Datei und setzen einen Kubectl-Apply-Befehl ab. Es dauert nur rund 60 Sekunden, bis Sie Ihre Seite mit vorangestelltem https:// erreichen. Wenn Sie der Browser auch nach Minuten noch mit einer Zertifikatswarnung begrüßt, klemmt irgendetwas. Besorgen Sie sich mit `kubectl get pods` den Namen des Traefik-Pods und schauen mit `kubectl logs`, was Traefik daran hindert, ein Zertifikat zu besorgen.

Verkehrssteuerung

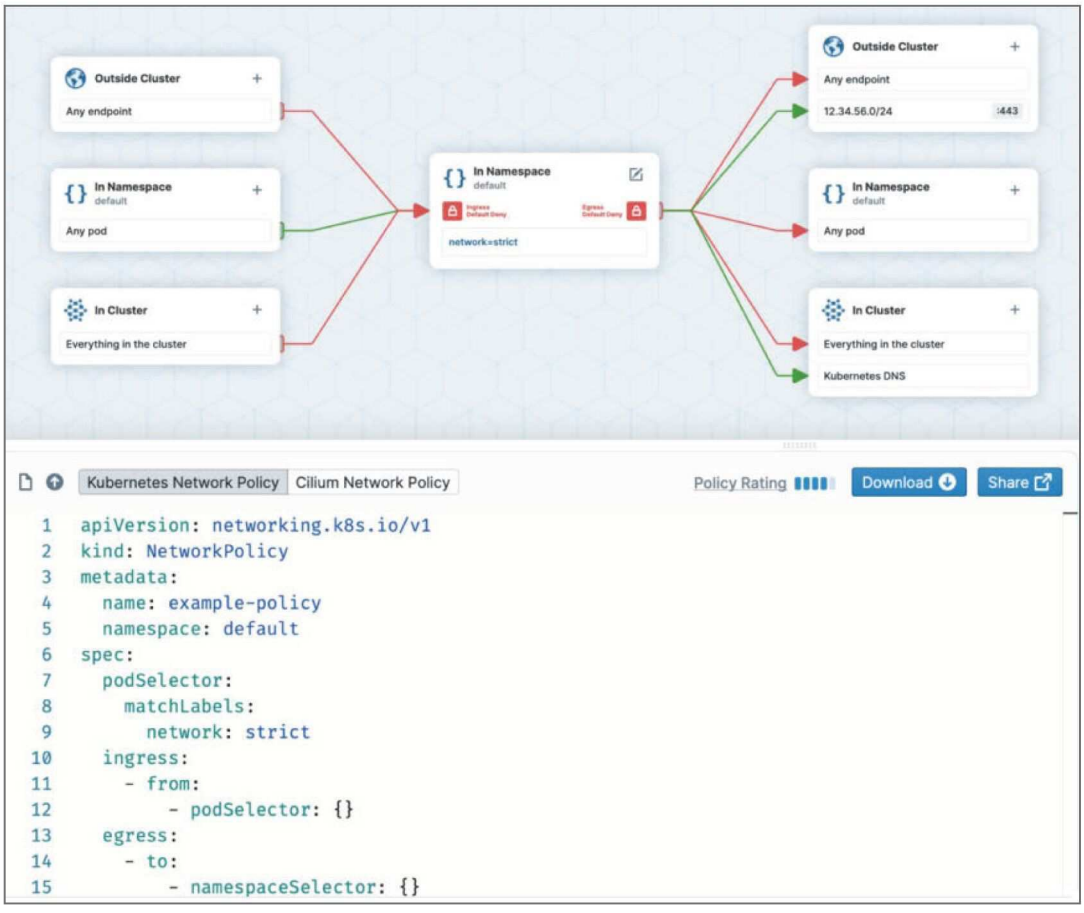
Das nächste Problem, das Sie auf dem Weg zu einem sichereren Cluster angehen sollten, ist der ungezügelte Netzwerkverkehr von Pods. Unternimmt man nichts, kann jeder Pod jeden Service im Cluster ansprechen, auch über Namespace-Grenzen hinweg. Außerdem kommt er ungehemmt ins Internet. In der Theorie ist das kein Problem, wenn man davon ausgeht, dass man als Administrator alle Pods selbst kontrolliert und weiß, was darin passiert. Doch mit dieser Einstellung macht man es Angreifern wahnsinnig leicht. Sobald die es schaffen, ihren Schadcode auf einem einzigen Pod auszuführen, können sie mühelos weiteren Code aus dem Internet nachladen und sich im Cluster oder gleich im ganzen Netzwerk ausbreiten. Das muss nicht sein.

Die allermeisten Pods brauchen keinen Weg ins Internet; bei Containern kann und sollte man da noch strenger sein als bei klassisch installierter Software. Bei letzterer gibt es manchmal noch das Szenario, dass sie regelmäßig einen Herstellerserver nach neuen Updates fragen müssen. Bei Containern fällt auch das weg, weil Software im Container sich nicht selbst aktualisieren soll. Das funktioniert in Containern anders: Gibt es Updates, wird der Container durch einen mit frischem Image ersetzt.

Nicht zuletzt durch die schwere Sicherheitslücke in der Java-Logging-Bibliothek Log4j haben viele Serverbetreiber erkannt, dass sie in der Vergangenheit zu großzügig mit dem Recht umgegangen sind, Kontakt mit dem Internet aufzunehmen. Der Log4-Shell-Angriff war darauf angewiesen, dass Log4J Code aus dem Internet nachlud.

Schränken Sie den Verkehr rigoros ein, damit es so weit nicht kommt. Dafür kennt Kubernetes das Konzept der NetworkPolicies, die wie Cluster-interne Firewallregeln funktionieren. Wie bei anderen Firewalls gibt es zwei Arten von Regeln; solche für eingehenden (Ingress) und solche für ausgehenden Verkehr (Egress). Eingerichtet werden die Policies, wie alles in Kubernetes, mit einer YAML-Definition, die wie die folgende aussieht:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: example-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
```



Mit dem Network-Policy-Editor von Cilium hat man passende Regeln ohne Tipperei schnell zusammengebaut.

```
    example.org/network-rule: strict
ingress:
  - from:
      - podSelector: {}
egress:
  - to:
      - podSelector:
          matchLabels:
            app: database
    ports:
      - port: 3306
  - to:
      - namespaceSelector: {}
        podSelector:
          matchLabels:
            k8s-app: kube-dns
    ports:
      - port: 53
        protocol: UDP
  - to:
      - ipBlock:
          cidr: 12.34.56.0/24
    ports:
      - port: 443
```

Wie jedes Objekt bekommt eine NetworkPolicy im Abschnitt `metadata:` einen Namen. Unter `spec:` folgt der Inhalt der Regel. Zunächst legt man fest, für welche Pods sie gelten soll. Zum Einsatz kommt ein `podSelector`, der zum Beispiel auch bei Services genutzt wird. Kubernetes sucht damit nach allen Pods, an die man ein bestimmtes Label angeheftet hat. Labels können Sie als Administrator grundsätzlich recht frei vergeben. Es ist jedoch ausdrücklich

empfohlen, selbst ausgedachte Labels mit einem eigenen Präfix in Form der eigenen Domain zu kennzeichnen. Dann kann man sichergehen, dass sie nicht mit Kubernetes-eigenen Labels kollidieren (auch nicht mit einem, das sich die Kubernetes-Entwickler in Zukunft noch ausdenken). Im Beispiel sollen alle Pods eine Regel bekommen, die mit dem Label `example.org/network-rule` und dem Wert `strict` markiert sind.

Es folgt eine recht kurze Ingress-Regel mit einem leeren Objekt, das Einschränkungen enthalten könnte, im Beispiel aber leer ist. Die Folge: Alle Pods im Cluster dürfen Pods mit dieser NetworkPolicy kontaktieren. Möchte man dagegen, dass kein anderer Pod auf einen Pod zugreifen darf, muss die Liste der Ingress-Regeln leer sein:

```
ingress: []
```

Ausgehende Regeln gibt es gleich drei. Die erste erlaubt Verkehr zu Pods mit dem Label `app: database`. Ohne die zweite Regel funktioniert sie aber nicht – ein typischer Fehler, mit dem sich NetworkPolicy-Einsteiger teils lange herumärgern. Die zweite Regel erlaubt dem Pod, den Kubernetes-internen DNS-Server auf Port 53 zu nutzen, um die Adresse der Datenbank aufzulösen. Ohne DNS geht nicht viel. Die dritte Egress-Regel ist ein Beispiel für externen Verkehr. Der Pod darf damit alle IP-Adressen zwischen 12.34.56.1 und 12.34.56.254 auf Port 443 ansprechen.

Die Syntax der NetworkPolicies ist zu Beginn etwas verwirrend und schnell hat man sich verkonfiguriert, was entweder zu unnötig laxen Regeln oder gescheiterten Verbindungen führt. Sehr empfehlenswert (nicht nur für Einsteiger) ist der Onlinegenerator von Cilium (editor.cilium.io), in dem man seine Regeln im Browser zusammenklickt und an roten und grünen Pfeilen sieht, was erlaubt und verboten ist. Ganz uneigennützig stellt Cilium den Editor nicht bereit: Cilium ist ein Open-Source-Projekt, das eine zusätzliche Netzwerkschicht für Kubernetes entwickelt, die unter anderem noch genauer filtern kann. Und so weist der Editor an einigen Stellen darauf hin, was mit Kubernetes-Bordmitteln (noch) nicht möglich ist und wie übersichtlich die Cilium-Syntax im Vergleich aussieht. Cilium ist reizvoll für große Organisationen, aber kein Projekt für Einsteiger.

In NetworkPolicies kann man viel Zeit investieren. Ein sicherer und effizienter Weg: Man verbietet zunächst mal alles und gibt dann gezielt die wenigen Ports und Verbindungen frei, die wirklich nötig sind. Wenn Sie so vorgehen, erfinden Sie passende NetworkPolicies bald genauso routiniert, wie Sie Deployments, Services und IngressRoutes anlegen. Ausgangspunkt für das Absicherungsprojekt ist eine strikte Regel, die alle Schotten dicht macht:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: default-deny
  namespace: default
spec:
  policyTypes:

  - Ingress
  - Egress
  podSelector: {}
  ingress: []
  egress: []
```

Weil eine NetworkPolicy immer zu einem Namespace gehört, braucht man diese Regel für jeden Namespace. Ist das erledigt, geht erst mal nichts mehr – damit das WordPress-Beispiel wieder läuft, brauchen Sie Regeln, die WordPress Kontakte zur Datenbank gestatten und Traefik Kontakt zu WordPress. Wenn Sie diese Herausforderung als Übungsaufgabe nutzen wollen, ein Tipp: Zum erfolgreichen Verbindungsaufbau gehören `ingress` und `egress`. Eine mögliche Lösung finden Sie im GitHub-Repository zum Artikel (siehe ct.de/w6xs) – es führen aber viele Wege zum Ziel.

Rechte verwalten

Beim Einrichten des Clusters im ersten Teil der Reihe hat K3S stumm einen Benutzer angelegt, der mit vollen Rechten ausgestattet ist. Er meldet sich mit einem Schlüsselpaar aus öffentlichem und privatem Schlüssel an – mit diesen Feinheiten mussten Sie sich bislang nicht beschäftigen, weil K3S die Erzeugung erledigt und alles in eine fertige YAML-Datei gelegt hat, die Sie als ~/.kube/config auf die lokale Maschine kopiert haben.

Solange Sie allein sind und auf dieses Schlüsselpaar aufpassen, ist das nicht grundsätzlich unsicher – auf Ihrer lokalen Maschine liegen ja oft auch SSH-Schlüssel für diverse Server. Problematisch wird der Kubernetes-Superuser aber spätestens, wenn Sie Ihren Cluster mit Kollegen teilen wollen. Denen wollen Sie nicht unbedingt Vollzugriff und Ihren einzigen Schlüssel geben. Hier kommt die sehr granulare

Rollenverwaltung von Kubernetes zum Zug. Um beim WordPress-Beispiel zu bleiben, könnten sich die Frontend-Entwickler einen Zugang wünschen, um im Namespace Frontend Änderungen an Pods vornehmen zu können. Sie wollen vielleicht mal das Image austauschen. Mit der IngressRoute und anderen Ressourcen haben sie aber nichts zu tun.

Dafür brauchen Sie zunächst zwei neue Ressourcen: eine Role und ein RoleBinding. Erstere definiert, was ein Rolleninhaber genau darf. Per RoleBinding wird die Rolle an einen Nutzer oder eine Nutzergruppe gebunden – den Nutzer mit dem Nutzernamen `frontend-guy` erzeugen Sie erst danach.

Folgender YAML-Block definiert die Rolle `frontend-dev` im Namespace `frontend` und berechtigt dazu, Pods zu erstellen und zu löschen:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: frontend
  name: frontend-dev
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs:
      [
        "create",
        "delete",
        "deletecollection",
        "get",
        "list",
        "patch",
        "update",
        "watch",
      ]
```

Das Beispiel zeigt, wie granular Sie Berechtigungen steuern dürfen. `apiGroups` enthält nur einen leeren String, damit gilt die Regel für Core-Kubernetes-Objekte, zu denen Pods gehören. Die `apiGroup` eines Kubernetes-Objekts erkennen Sie mit einem Blick auf die Angabe `apiVersion` in der YAML-Datei. Die oben vorgestellte NetworkPolicy gehört beispielsweise zur Gruppe `networking.k8s.io`.

Unter `resources:` geben Sie eine Liste mit Objekten an, mit denen interagiert werden darf. Die Frontend-Kollegen im Beispiel müssen sich mit Pods begnügen. Dafür dürfen sie mit Pods in ihrem Namespace alles anstellen – freigegeben sind alle `verbs`, die Kubernetes kennt. Diese Liste dient hier nur der Anschauung, anstatt sie auszuschreiben, würde man in der Praxis die Wildcard `[*]` nutzen. Per Wildcard könnten Sie zum Beispiel auch eine Rolle bauen, die in einem Namespace alle Ressourcen sehen darf.

Damit der Benutzer `frontend-guy` diese Rolle bekommt, brauchen Sie noch eine Zuweisung:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: frontend-dev
  namespace: frontend
subjects:
  - kind: User
    name: frontend-guy
    apiGroup: rbac.authorization.k8s.io
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: frontend-dev
```

In diesem Objekt werden Benutzer und Rolle miteinander verknüpft. Legen Sie eine Datei mit den beiden Objekten an und bringen sie per Kubectl-Apply in den Cluster.

Benutzer backen

Zum Erfolg fehlt nur noch der Benutzer `frontend-guy` selbst – doch einen Befehl wie `kubectl get users` oder eine User-Ressource, für die Sie eine YAML-Datei anlegen könnten, sucht man in der Kubernetes-Dokumentation vergeblich. So etwas kennt Kubernetes nicht, weil das System überhaupt keine Benutzer in seiner Datenhaltung speichert. Stattdessen arbeitet das Kubernetes-API nur mit Zertifikaten: Zum „Anlegen" eines Nutzers erzeugen Sie lokal ein Zertifikat mit einem öffentlichen und privaten Schlüssel und schreiben den Benutzernamen hinein. Dieses Zertifikat lassen Sie von der Zertifizierungsstelle von Kubernetes signieren. Diese Zertifizierungsstelle bringt Kubernetes immer mit und nutzt sie intern auch für andere Aufgaben, meist bekommen Sie davon nicht viel mit.

Mithilfe des privaten Schlüssels und des Zertifikats können Sie sich fortan ausweisen. Auch wenn der Benutzername nicht im Cluster gespeichert wurde, klappen Authentifizierung (Anmeldung) und Autorisierung (Rechtevergabe). Das Kubernetes-API weiß: „Ich habe dieses Zertifikat signiert, also kann ich den Angaben darin trauen".

Das Prinzip leuchtet ein, wenn man es einmal durchgespielt hat. Wenn Sie die Befehle nicht abtippen wollen, finden Sie sie ebenfalls im Repository. Die Reise beginnt auf einer lokalen Maschine mit dem Erzeugen eines RSA-Schlüsselpaars. Dafür kommt das klassische Werkzeug `openssl` zum Einsatz. Es funktioniert mittlerweile auch unter Windows, wir empfehlen Windows-Nutzern dennoch, die folgenden Schritte unter Linux oder im WSL nachzuspielen. Alle Dateinamen im folgenden Beispiel können Sie frei vergeben:

```
openssl genrsa -out user.pem
```

Damit liegt das Schlüsselpaar bereit, daraus wird jetzt eine Zertifikatsbestellung (Certificate Signing Request, CSR):

```
openssl req -new -key user.pem -out↵
↳ user.csr -subj "/CN=frontend-guy"
```

In dieser Zeile (und nirgends sonst) wird der Nutzername in der von LDAP bekannten Syntax mit vorangestelltem `CN=` (für Common Name) festgelegt. Heraus kommt eine Datei namens user.csr, die Sie direkt in Base64 wandeln müssen:

```
cat user.csr | base64
```

Die erzeugte Zeichenkette gehört in eine YAML-Datei, die einen CertificateSigningRequest für die Kubernetes-Zertifizierungsstelle definiert:

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: csr-frontend-guy
spec:
  groups:
  - system:authenticated
  request: <Base64-String>
  signerName: ↵
  ↳kubernetes.io/kube-apiserver-client
  expirationSeconds: 157680000
  usages:
  - digital signature
  - key encipherment
  - client auth
```

Den Namen des Objekts dürfen Sie frei wählen und auch die `expirationSeconds` können Sie festlegen. Das Beispiel-Zertifikat gilt knapp 5 Jahre. Bereiten Sie diesen YAML-Schnipsel vor und bringen ihn in den Cluster:

```
kubectl certificate ↵
↳approve csr-frontentd-guy
```

Weil Sie ja momentan mit dem Superuser arbeiten, dürfen Sie Ihre eigene Anfrage direkt genehmigen. Im Cluster liegt jetzt ein signiertes Zertifikat, interessant ist dessen öffentlicher Schlüssel. Den bekommen Sie per Kubectl-Befehl:

```
kubectl get csr csr-frontend-guy ↵
↳ -o jsonpath='{.status.certificate}'
```

Der zugehörige private Schlüssel hat die ganze Zeit auf der lokalen Platte auf seinen Einsatz gewartet – `openssl` hat ihn zu Beginn in die Datei user.pem gelegt, er hat den Computer aber nie verlassen. Das soll auch so bleiben. Auch ihn müssen Sie sich einmal als Base64-String anzeigen:

```
cat user.pem | base64
```

Jetzt haben Sie alle wichtigen Bausteine zusammen, mit denen sich der Frontend-Entwickler namens `frontend-guy` anmelden kann. Er muss beide Base64-Zeichenketten in seine Datei ~/.kube/config unterhalb von `users:` einbauen, etwa so:

```
users:
  frontend-guy:
    client-certificate-data: <Pub Key>
    client-key-data: <Private Key>
```

Beispiele und Dokumentation

ct.de/w6xs

Wenn Sie ausprobieren wollen, was der Beispielnutzer `frontend-guy` alles darf, bauen Sie diesen Block zusätzlich in Ihre Konfiguration ein (ohne die Zugangsdaten für Ihren Superuser zu löschen) und verweisen im Kontext auf diesen Nutzer. Der Aufruf `kubectl get pods` sollte eine Fehlermeldung auslösen, nur die Abfrage der Pods im Namespace `frontend` darf zum Erfolg führen:

```
kubectl get pods -n frontend
```

Wenn Sie Administrator in einer größeren Organisation sind, wollen Sie die Schritte sicher automatisieren, um zügiger Zertifikate für das Team auszustellen. Die Schlüsselerzeugung mit `openssl` sollte bestenfalls auf der Entwicklermaschine des jeweiligen Nutzers passieren – den privaten Key müssen Sie als Administrator nicht kennen. Das ist der Reiz von asymmetrischer Kryptografie.

Zwischenfazit

Mit TLS und gültigem Zertifikat haben Sie Ihren Besuchern einen sicheren Weg zu Ihren Diensten bereitet, mit NetworkPolicies die Pods im Cluster voneinander abgeschottet und mit einem beschnittenen Account Ihre Kollegen gezielt berechtigt. Damit ist es um die Sicherheit im Cluster deutlich besser bestellt, der Kubernetes-Werkzeugkasten ist aber noch längst nicht ausgeschöpft und es gibt noch andere Sicherheitsmechanismen zu entdecken – eine Aufgabe für Fortgeschrittene ist beispielsweise die Anbindung von SELinux. Bevor Sie sich an diese Baustelle machen, ist es jetzt aber Zeit für den vergnüglichen Teil: Der Cluster ist bereit für richtige Anwendungen mit Speicherplatz und TLS, die Sie im nachfolgenden Artikel kennenlernen. (jam) ct

Redundanter Speicher mit Longhorn

In einem Kubernetes-Cluster laufen Anwendungen skalierbar und redundant. Damit auch die anfallenden Daten auf mehreren Servern liegen und der Speicherplatz mit den Anforderungen wächst, braucht man eine Erweiterung wie Longhorn. Sie macht Volumes redundant – eine Backup-Strategie gibt es obendrauf.

Von **Jan Mahn**

Bild: KI Midjourney | Bearbeitung: c't

Mit dem Umstieg von einem einzelnen Docker-Server auf einen Kubernetes-Cluster eröffnen sich schier grenzenlose Möglichkeiten, die eigene Anwendung zu skalieren. Wie Sie diesen Weg beschreiten und vom Docker- zum Kubernetes-Kenner werden, haben wir auf den vorangegangenen Seiten im Kapitel „Der Lernpfad zum Kubernetes-Kenner“ ab S. 38 beschrieben. Wachsen die Anforderungen, kann man mit Kubernetes problemlos Server nachbestellen und in den Cluster aufnehmen, um größeren Lasten zu begegnen. Was mit Kubernetes-Bordmitteln aber nicht mitwächst, ist der Speicherplatz. Muss ein Container etwas auf der Festplatte speichern, geschieht das über mehrere Abstraktionsschichten (VolumeMount, Volume, PersistentVolumeClaim und StorageClass, siehe Artikel „Volumes, Secrets und ConfigMaps“ auf S. 60). Der Prozess, der im Container läuft, bekommt von solchen Details nichts mit – ihm setzt die Container-Runtime ein Dateisystem vor, das er lesen und auf Wunsch auch beschreiben kann.

Ein ganz einfacher Anbieter von Kubernetes-Speicherplatz ist zum Beispiel der LocalPathProvisioner von Rancher (siehe ct.de/w6tu), den die leichtgewichtige Kubernetes-Distribution k3s bereits mitliefert. Der schnappt sich einfach einen Ordner im Dateisystem des Nodes, auf dem der Container läuft und reicht ihn an den Container weiter. Dieses Verhalten entspricht ziemlich genau dem, was Docker-Nutzer von „named volumes“ kennen, also solchen Volumes, die man mit Befehlen wie `docker volume ls` und `docker volume create` verwaltet. Doch in einem Cluster ist das gar nicht mal so praktisch: Liegen die Daten auf einem einzigen Node, wird es zur Qual, den Pod auf einen anderen Node umziehen zu lassen – fällt ein Node mit angehängtem Volume aus, kann Kubernetes ihn nicht woanders unterbringen. Redundant gespeichert wird vom LocalPathProvisioner auch nichts.

Auftritt von Longhorn

Viel mehr Komfort und redundanten Speicherplatz bietet die Open-Source-Software Longhorn. Longhorn kann Daten automatisch in mehreren verteilt gespeicherten Kopien, auch Replicas genannt, auf dem gleichen Stand halten. Ganz nebenbei bekommt man mit Longhorn eine grafische Oberfläche für die Verwaltung sowie einen Backup-Mechanismus, der auf externe Ziele sichert. Die Einrichtung von Longhorn in einem bestehenden Cluster ist vergleichsweise schnell erledigt, ein paar Tücken warten dann im täglichen Betrieb. Dass Anbieter wie Longhorn eine solche Speicherschicht für Kubernetes bauen können, liegt an einem Konzept namens „Container Storage Interface“ (CSI). Das ist die Kubernetes-Plug-in-Schnittstelle für Speicherplatz (sogenannten Block-Storage). Die wurde von den Kubernetes-Maintainern geschaffen, als sich immer klarer abzeichnete, dass man verschiedene Speicheranbindungen unmöglich im Kubernetes-Code selbst entwickeln kann.

Longhorn installieren Sie am schnellsten über den Kubernetes-Paketmanager Helm, der auf der lokalen Entwicklermaschine eingerichtet ist und dort auf die Kubeconfig-Datei mit den Cluster-Zugangsdaten zugreift (siehe Artikel „Services und Ingress mit Traefik“ auf S. 52). Vor dem Helm-Einsatz müssen Sie die Kubernetes-Welt aber kurz verlassen und auf den Servern zwei Pakete auf Linux-Ebene installieren. Longhorn erwartet, dass iSCSI auf allen Maschinen installiert ist. Unter Debian und Ubuntu erreichen Sie das mit dem folgenden Befehl, direkt auf den Servern ausgeführt:

```
sudo apt install open-iscsi
```

Unter SUSE und openSUSE mit dem Paketmanager Zypper heißt das Paket ebenfalls `open-iscsi`. Die mehrschrittige Anleitung für Server-Distributionen aus der Red-Hat-Großfamilie finden Sie in der Longhorn-Dokumentation (siehe ct.de/w6tu).

Zweite Voraussetzung ist ein NFS-Client. Den bekommen Sie unter Debian und Ubuntu per

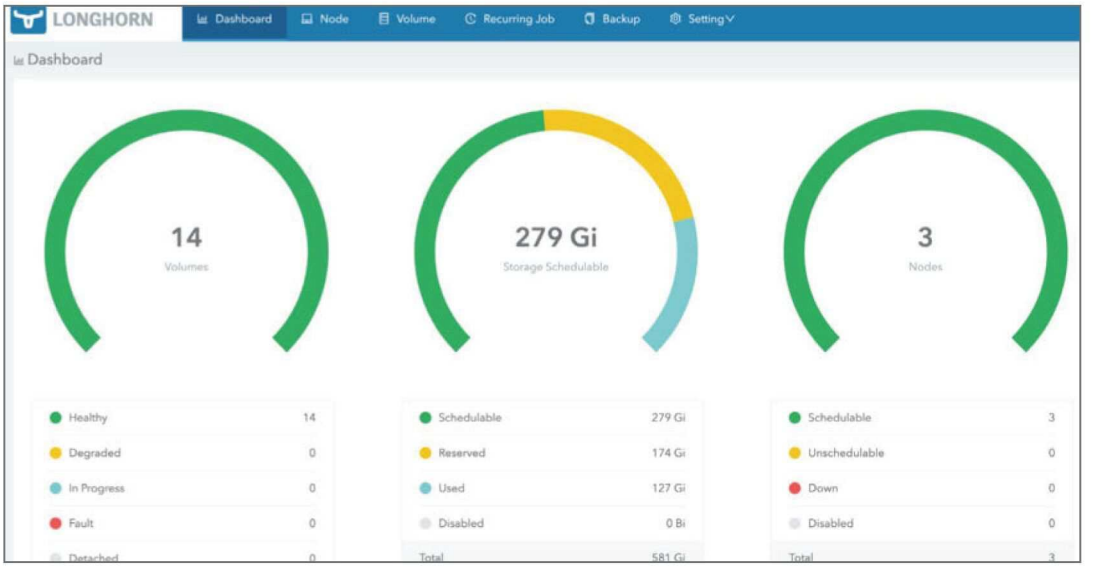
```
sudo apt install nfs-common
```

Unter Red Hat heißt das Paket `nfs-utils`, bei SUSE `nfs-client`. Wenn Sie von manuellen Paketinstallationen genervt sind, sollten Sie die Installation der beiden Pakete für all Ihre Kubernetes-Server automatisieren – zum Beispiel mit Ansible oder zumindest per Skript.

Sind diese Linux-Vorarbeiten erledigt, ist es Zeit für den Kubernetes-Paketmanager Helm und die Longhorn-Installation selbst. Zunächst müssen Sie die Paketquelle von Longhorn einbinden:

```
helm repo add longhorn ↵
↳https://charts.longhorn.io
helm repo update
```

Anschließend ist Longhorn mit einem Befehl im Namespace longhorn-system installiert. Wenn es den



Speicherplatz im Blick: Longhorn stellt Informationen zu Servern und Volumes in einer Weboberfläche zur Verfügung.

vor der Installation noch nicht gibt, wird er direkt angelegt. Die Longhorn-Entwickler empfehlen, diesen Namespace nicht zu ändern. Die Installation starten Sie mit:

```
helm install longhorn ↵
↳longhorn/longhorn ↵
↳--namespace longhorn-system ↵
↳--create-namespace --version 1.4.1
```

Wenn Sie diesen Artikel deutlich nach Mai 2023 lesen, sollten Sie mit dem Befehl `helm search repo longhorn` nach der aktuellen Version Ausschau halten und diese Angabe im Befehl anpassen. Alle Befehle aus diesem Artikel finden Sie auch über ct.de/w6tu zum Kopieren.

Nach der Installation verrät der folgende Befehl, ob alle Pods für Longhorn einsatzbereit sind:

```
kubectl get pods -n longhorn-system
```

Solange dort nicht hinter jeder Zeile Running steht, muss sich die Software noch berappeln. Anhand der Pod-Namen kann man erahnen, was Longhorn im Hintergrund anstellen muss, um redundanten Speicherplatz bereitzustellen: Es gibt Attacher, Provisioner, Resizer, Snapshotter und Manager, die für verschiedene Abschnitte im Lebenszyklus eines Volumes zuständig sind. Als Longhorn-Anwender hat man mit diesen Pods wenig zu tun, weil sie die meisten Schritte automatisch erledigen.

Für die wenigen Aufgaben, die man als Admin überhaupt per Hand anschieben muss, gibt es eine Longhorn-Weboberfläche. Zwei Pods, die longhorn-ui im Namen tragen, liegen dafür nach der Installation bereit und auch ein Service ist eingerichtet. Damit man von außen darauf zugreifen kann, muss man eingehenden HTTP-Verkehr auf diesen Service umleiten. Dafür gibt es gleich zwei Wege, einen permanenten und einen temporären.

Wer nur ab und zu auf die Oberfläche schauen möchte, greift zum Kubernetes-Werkzeug Port-Forward. Damit kann man sich als Administrator eine direkte Verbindung zwischen einem Port des eigenen Rechners und einem Service im Cluster aufbauen – auch abseits von Longhorn eine nützliche Funktion und auch nicht auf HTTP beschränkt. Die Longhorn-Oberfläche gelangt mit folgendem Befehl auf den eigenen Computer:

```
kubectl --namespace longhorn-system ↵
↳port-forward ↵
↳service/longhorn-frontend 3080:80
```

Solange die Kommandozeilensitzung geöffnet ist, erreichen Sie die Longhorn-Oberfläche im Browser

unter http://localhost:3080. Zu sehen gibt es zu Beginn noch nicht viel, weil keine Volumes angelegt sind, die verwaltet werden müssen. Wie Sie Ihr erstes Longhorn-Volume erzeugen, lesen Sie im nächsten Abschnitt „Ein Volume, bitte“. Wenn Sie für die Administration des Clusters das GUI von Lens nutzen (zum Download über ct.de/w6tu), können Sie eine solche Portweiterleitung dort mit einem Klick in der Oberfläche aktivieren.

Reichen Ihnen die gelegentlichen Weiterleitungen per Port-Forward nicht, können Sie die Oberfläche auch über die IngressRoute eines Reverse-Proxy dauerhaft anbinden. Doch Achtung: Von Haus aus ist keine Authentifizierung vorgesehen. Auch diese Aufgabe müssen Sie also einem Reverse-Proxy überlassen, weil Sie die Verwaltung Ihrer Anwendungsdaten sicher nicht ohne Zugriffsschutz ins Internet hängen wollen.

Sollten Sie die Open-Source-Anwendung Traefik als Reverse-Proxy im Einsatz haben (wie im Artikel „Services und Ingress mit Traefik“ auf S. 52 beschrieben), ist die Route recht schnell angelegt und mit HTTP-Basic-Auth abgesichert. Im Kasten „IngressRoute mit Anmeldung“ finden Sie die nötigen Schritte. Voraussetzung ist, dass Sie TLS mit einem Zertifikat eingerichtet haben – ohne Transportverschlüsselung ist Basic-Auth kein sicherer Zugriffsschutz.

Ein Volume, bitte

Damit in der Weboberfläche etwas passiert, brauchen Sie ein oder mehrere Volumes. Ein Volume kann man – wie in der Kubernetes-Welt üblich – auf unterschiedlichen Wegen erzeugen. Der komfortabelste führt über einen PersistentVolumeClaim (PVC), also ein Kubernetes-Objekt, das man separat anlegt und das ein Volume vorbestellt. Ein solcher PVC wird dann an einen Pod geheftet, in dessen Konfiguration man auch festlegt, welcher Pfad eines Containers im Volume landen soll. Um Longhorn für ein Volume zu nutzen, muss man bei der Definition des PVC nur eine StorageClass angeben, die Longhorn verwendet. Bei der Installation legt Longhorn bereits eine solche StorageClass namens `longhorn` an.

Als simples Beispiel reicht ein Nginx-Webserver, der seine Website in einem Volume ablegen soll. Zunächst brauchen Sie ein PVC namens `pvc-nginx`:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-nginx
spec:
  accessModes:
   - ReadWriteOnce
  volumeMode: Block
  storageClassName: longhorn
  resources:
    requests:
      storage: 1Gi
```

Damit wird 1 GByte aus der storageClass `longhorn` vorbestellt. Der `accessMode` ist ein nicht unwichtiges Detail. `ReadWriteOnce` ist die Standardeinstellung und auch die, die Sie in den allermeisten Fällen nutzen wollen. In diesem Modus darf genau ein Pod lesend und schreibend darauf zugreifen.

Longhorn kennt auch einen zweiten Modus: `ReadWriteMany` vollbringt das Kunststück, ein Laufwerk

Ein Volume, drei Replicas. Longhorn hält die Daten auf drei Servern vor.

IngressRoute mit Anmeldung

Traefik kann eingehende Anfragen gleich mit mehreren Authentifizierungsverfahren vor unbefugten Zugriffen schützen. In komplexeren Umgebungen mit vielen Nutzern kann Traefik die Authentifizierung auch an andere Server delegieren, für eine Admin-Seite ist HTTP-Basic-Auth das einfachste Verfahren: Bei der Einrichtung hinterlegt man eine Kombination aus Benutzername und Hash eines Kennworts im Cluster. Möchte man sich mit der Seite verbinden, fragt der Browser die Anmeldedaten in einem schmucklosen Fenster ab.

Los geht die Einrichtung auf einer Maschine mit Linux, macOS oder dem WSL unter Windows. Mit der folgenden Zeile bauen Sie einen Base64-enkodierten String zusammen, der Benutzername und gehashtes Kennwort im richtigen Format enthält. `htpasswd` stammt aus dem Apache-Universum und Traefik nutzt dessen etabliertes Format:

```
htpasswd -nb admin secretPw |↵
↳ openssl base64
```

Ersetzen Sie den Benutzernamen `admin` und das Passwort durch eigene Daten. Die Zeichenkette kommt dann in ein neues Kubernetes-Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: longhorn-ui-credentials
  namespace: longhorn-system
data:
  users: |
    YWRtaW46JGFwcjEkSVdYUnZWQUkkbOd...
```

Das zweite Kubernetes-Objekt ist eine Middleware – das ist Traefiks Konzept, um in jeglichen Verkehr einzugreifen. Diese Middleware namens `longhorn-auth` verweist auf das oben angelegte Secret:

```
apiVersion: traefik.containo.us/v1alpha1
kind: Middleware
metadata:
  name: longhorn-auth
  namespace: longhorn-system
spec:
  basicAuth:
    secret: longhorn-ui-credentials
```

Die Longhorn-Oberfläche soll unter der Adresse example.org/longhorn veröffentlicht werden. Damit das klappt, braucht es noch eine weitere Middleware, die eine Schwäche der Longhorn-Oberfläche ausgleicht. Die ist von Haus aus nicht darauf vorbereitet, unterhalb eines Pfads wie /longhorn erreichbar zu sein. Per stripPrefix-Middleware kann Traefik diesen Pfadbestandteil aus den Anfragen herausschneiden. Ein Handgriff, den Sie auch bei vielen anderen Diensten kennen sollten, die Sie unter einem Pfad bereitstellen wollen:

```
apiVersion: traefik.containo.us/v1alpha1
kind: Middleware
metadata:
  name: longhorn-strip
  namespace: longhorn-system
spec:
  stripPrefix:
    prefixes:
      - /longhorn
```

mehreren Pods auf mehreren Servern anzubieten, die alle gleichzeitig lesen und schreiben dürfen. Dafür gibt es bei einigen Anwendungen gute Gründe, alle Probleme kann man damit aber nicht lösen – dazu später mehr.

Wenn Sie den PVC in den Cluster befördert haben, können Sie schon mal einen Blick auf die Weboberfläche werfen. Unter dem Reiter Volume (oben in der blauen Leiste) taucht der Eintrag auf, der Status steht zunächst auf Detached – und zwar so lange, bis der erste Pod damit verbunden wird. Als Beispiel dient ein einfacher Nginx-Server:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-example
  namespace: default
spec:
  containers:
```

Der dritte Schritt ist weitgehend Standardkost: Sie brauchen eine IngressRoute, die auf den Service für die Traefik-Oberfläche zeigt und die mit den Middlewares verknüpft ist. In einer YAML-Definition sieht diese Kombination so aus. Die Domain müssen Sie an Ihre eigene Umgebung anpassen:

```
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: longhorn-ingress
  namespace: longhorn-system
spec:
  entryPoints:
   - websecure
  routes:
   - match: Host(`www.example.org`) ↵
          ↳&& PathPrefix(`/longhorn`)
     kind: Rule
     services:
      - name: longhorn-frontend
        port: 80
     middlewares:
      - name: longhorn-auth
        namespace: longhorn-system
      - name: longhorn-strip
        namespace: longhorn-system
```

Diese vier Objekte speichern Sie am besten per --- getrennt in einer Datei und bringen sie gemeinsam per Kubectl-Befehl in den Cluster. Direkt im Anschluss ist Longhorn unter, gesichert mit Basic-Authentifizierung und dem zuvor festgelegten Kennwort, www.example.org/longhorn/ erreichbar. Einziger kleiner Schönheitsfehler: Der / am Ende ist Pflicht, sonst lädt die Seite nicht.

```
   - name: nginx
     image: nginx:alpine
     imagePullPolicy: IfNotPresent
     ports:
      - containerPort: 80
     volumeMounts:
      - name: nginx-vol
        mountPath: /usr/share/nginx
  volumes:
   - name: nginx-vol
     persistentVolumeClaim:
       claimName: pvc-nginx
```

Wenn Sie auch Nginx in den Cluster gebracht haben, wechselt der Status des Volumes in der Oberfläche. Sofern Sie einen Cluster aus mindestens drei Servern haben, sollte in der ersten Spalte in grüner Schrift Healthy stehen. Ein Klick auf den Namen zeigt eine Detailansicht mit den Replicas.

Sollten Sie das Beispiel auf einem Single-Node-Cluster ausprobieren, schafft es das Volume nicht in den Healthy-Zustand und bleibt Degraded. Das ist kein Fehler, sondern eine logische Folge der Standard-StorageClass namens `longhorn`. Die ist so konfiguriert, dass ein Volume immer in drei Replicas vorgehalten wird und diese auf drei Nodes verteilt sind. Gibt es nur einen Node, kann Longhorn die beiden Kopien nicht platzieren und bezeichnet das Volume als Degraded (selbiges passiert auch, wenn mal ein Node ausfällt).

Wenn Sie Longhorn-Volumes brauchen, die nicht repliziert werden sollen, müssen Sie eine eigene StorageClass dafür anlegen. Auch das ist kein großer Akt und mit einem Kubernetes-Objekt erledigt:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: longhorn-single
provisioner: driver.longhorn.io
allowVolumeExpansion: true
reclaimPolicy: Delete
volumeBindingMode: Immediate
parameters:
  numberOfReplicas: "1"
  staleReplicaTimeout: "2880"
  fromBackup: ""
  fsType: "ext4"
```

Entscheidende Abweichung ist die Zeile `numberOfReplicas: "1"`. Wenn Sie sich für die weiteren Konfigurationsmöglichkeiten interessieren, finden Sie alle Parameter in der Longhorn-Dokumentation (siehe ct.de/w6tu).

Überall Nägel

Wer einen Hammer hat, neigt dazu, in jedem Problem einen Nagel zu sehen. Diese Weisheit gilt auch für Longhorn, das man leicht mit einem Universalwerkzeug für alle redundanten Speicheraufgaben verwechseln kann. Schnell hat man zum Beispiel

eine MariaDB- oder PostgreSQL-Datenbank per Kubernetes-Deployment mehrfach hochgefahren und per Longhorn ein gemeinsames Longhorn-Volume im Modus ReadWriteMany angehängt. Das Ergebnis: eine korrupte Datenbank und mehrere abstürzende Pods. Schuld am Datensalat ist aber nicht Longhorn, sondern die Funktionsweise von Datenbanksoftware. Die ist schlicht nicht darauf ausgelegt, dass mehrere unabhängige Datenbankprozesse auf einen Datensatz schreiben.

Eine SQL- oder NoSQL-Datenbank lässt sich also leider nicht so einfach skalieren: Das funktioniert ausschließlich auf Ebene der Datenbankverwaltung selbst, nicht auf Ebene des Festplattenspeichers. Jede Datenbank handhabt den Clusterbetrieb ein bisschen anders. Bei MariaDB heißt die Funktion „MariaDB Galera Cluster“, für PostgreSQL gibt es PgCluster und für MongoDB Atlas. Lohnen kann sich auch ein Blick auf jüngere Datenbanken, die von Anfang an auf den Kubernetes-Einsatz ausgelegt wurden. Die Datenbank CockroachDB zum Beispiel verhält sich gegenüber dem Client (mit einigen Einschränkungen) wie eine PostgreSQL-Datenbank, läuft aber von Haus aus im Clusterbetrieb.

In allen clusterbaren Datenbanken laufen mehrere Pods mit demselben Container-Image, jeder Pod hat aber ein eigenes Volume. Ein Algorithmus wie Raft (siehe Artikel „Verteilte Systeme mit Raft-Algorithmus“ auf Seite 98) erledigt die Replikation. Keine gute Idee ist es, eine solche Datenbank zu nutzen und jeden Datenbank-Pod auf ein eigenes Longhorn-Volume mit aktivierter Replikation schreiben zu lassen. Die Daten werden dann unnötig und auf Kosten der Performance gleich mehrfach dupliziert, wie das folgende Beispiel ganz konkret zeigt: Eine Datenbank mit eingebauter Replikation läuft in drei Pods (db-0, db-1 und db-2), verteilt auf drei Maschinen. Gibt man jedem Pod jetzt ein replizierendes Volume, kopiert Longhorn insgesamt 3×3 Replicas mit der gesamten Datenbank über die drei Maschinen – ein zuverlässiger Weg, um den Cluster ganz ohne Benutzer auszulasten und auf Dauer in die Knie zu zwingen.

Wann immer eine Software selbst einen Replikationsmechanismus mitbringt, brauchen Sie eine StorageClass ohne Replikation wie die oben angelegte `longhorn-single`. Im Prinzip könnten Sie in solchen Fällen Longhorn einfach links liegen lassen und zum Beispiel wieder zum LocalPathProvisioner greifen. Doch damit verlieren Sie auch die anderen Vorteile – den schnellen Überblick per Weboberfläche und die externen Backups zum Beispiel.

Löschblockade

Volumes und PersistentVolumeClaims verhalten sich in vielen Punkten anders als andere Kubernetes-Objekte. Das liegt daran, dass sie nicht so flüchtig sind wie zum Beispiel Pods, die man einfach aus einem Image neu erzeugen kann. Daher kann man nicht alle Eigenschaften von PVCs ändern und sie auch nicht so leicht löschen. Wenn Sie es versuchen, beschwert sich das Kubernetes-API.

Ein Volume können Sie über die Weboberfläche löschen, indem Sie rechts auf das Menü klicken und Delete auswählen. Doch damit verschwindet der zugehörige PVC noch nicht. Der Befehl

```
kubectl get pvc
```

enthüllt: Der PVC steht weiter in der Liste, auch Stunden später noch. Nur sein Status hat sich geändert auf „Terminating“. Schuld daran ist ein Finalizer. Das ist ein Eintrag in den Metadaten eines Objekts, der Kubernetes am Löschen hindern kann. Wenn Sie sich sicher sind, dass Sie den PVC wirklich löschen möchten, entfernen Sie ihn per Kubectl. Für den Befehl brauchen Sie den Namen des PVC, nicht etwa des zugehörigen Volumes:

```
kubectl patch pvc <Name des PVC> ↵
↪-p '{"metadata":{"finalizers":null}}'
```

Im Alltag

In der Praxis kommt es häufiger vor, dass Sie den PersistentVolumeClaim nicht selbst anlegen, sondern Speicherplatz für eine Anwendung brauchen, die jemand anderes bereitstellt – zum Beispiel über ein Helm-Chart. Typischerweise legen die Autoren solcher Charts die Angabe einer StorageClass in die Values-Datei, sodass Sie vor der Installation nur in der jeweiligen Dokumentation nachschlagen müssen, unter welchem Schlüssel Sie den Namen Ihrer StorageClass eintragen müssen. In solchen Fällen ist es Ihre Verantwortung als Administrator, eine kluge Wahl zu treffen. In jedem Fall sollten Sie sich vorab informieren, was die Anwendung zu speichern gedenkt und ob es sich um eine Software handelt, die schon einen eigenen Replikationsmechanismus mitbringt.

Mit diesen Informationen sind Sie bereit, eigene Erfahrungen mit Longhorn zu sammeln. Mehr als nur einen Gedanken sollten Sie auf die Themen Snapshots und Backups verwenden. Ein Snapshot

ist eine Momentaufnahme des Inhalts – Longhorn zieht zum vorgegebenen Zeitpunkt eine Schicht in seiner Datenhaltung ein und schreibt alle Veränderungen seit dem letzten Snapshot in die neue Schicht. Das aktuelle Laufwerk ist eine Aneinanderreihung dieser Zwischenstände, die übereinandergelegt werden. Das Verfahren ähnelt der Funktionsweise von Container-Images, die auch aus Schichten bestehen. Einen Snapshot erstellen Sie, indem Sie die Detail-Seite eines Volumes anlegen und etwas nach unten scrollen.

Einen solchen Snapshot wieder einzuspielen, also in der Zeit zurückzureisen, ist kaum aufwendiger. Zunächst geht man in die Volume-Übersicht zurück und öffnet dort das Menü auf dem Volume. Dort wählt man den Befehl Detach und direkt im Anschluss wieder Attach – das Volume muss im Maintenance-Modus auf dem Node eingebunden werden. Anschließend wechselt man wieder auf die Detail-Seite, scrollt bis zu den Snapshots, klickt einen an und wählt den Menüpunkt Revert. Im Anschluss wieder detachen und ohne Maintenance-Modus wieder einhängen. Aber Achtung: Was für replizierende Volumes gilt, gilt auch für Snapshots: In Verbindung mit replizierenden Datenbanken können Sie sich fiese Probleme einfangen, wenn Sie plötzlich auf einen älteren Zeitpunkt springen. Nutzen Sie also bevorzugt die Backup- und Restore-Funktionen der Datenbank selbst.

Ein Backup folgt immer aus einem Snapshot. Konkret wird also zuerst eine Momentaufnahme erstellt und diese dann auf ein externes Backup-Ziel kopiert. Als Ablageort für Backups kennt Longhorn S3-Speicherplatz und NFS. Ersteren kann man bei zahlreichen Cloudprovidern anmieten, letzterer steht in vielen Unternehmensnetzen schon bereit. Die Einrichtung setzt etwas NFS- oder S3-Erfahrung voraus, ist dann aber mal wieder eine Kubernetes-Standard-Übung.

Im Kern muss man nur ein Secret mit den Zugangsdaten für den Backupplatz in den Cluster bringen und anschließend die Adresse des Speicherorts in der Longhorn-Konfiguration hinterlegen. Die findet man über die Reiter oben hinter dem Menüpunkt Setting/General. Wie Sie S3 oder NFS als Backupziel anbinden, würde den Rahmen dieser Einführung sprengen – die Longhorn-Dokumentation erklärt das sehr ausführlich. Dort gibt es rund um Backups und Snapshots noch mehr zu entdecken. Unter anderem erfahren Sie, wie Sie direkt an der StorageClass definieren, wie oft die Volumes ins Backup sollen.

Dokumentation
ct.de/w6tu

Das letzte Problem

Bevor Sie echte Daten in Longhorn verwalten, sollten Sie ausführliche Trockenübungen mit einem Experimentiercluster (am besten mit mindestens drei Nodes) vollziehen. Spielen Sie die möglichen Katastrophenfälle am besten alle durch und halten Sie die nötigen Schritte zum Einspielen von Backups in einem Handbuch für sich und eventuelle Admin-Kollegen fest. Zu diesen Übungen gehören Sprünge zu älteren Snapshots sowie das Einspielen von ganzen Backups aus dem S3- oder NFS-Speicher.

Eine weitere unerlässliche Übung: einen der Nodes ohne Vorwarnung abschalten (wie es bei einem Stromausfall passieren kann). Dabei können Sie auf ein ziemlich hinderliches Problem stoßen: Ein mit einem Longhorn-Volume verbundener Pod, der aus einem Deployment erzeugt wurde, verschwindet schlagartig (weil sein Node nicht mehr aufzufinden ist). Doch entgegen Ihrer Erwartung passiert nichts. Kubernetes unternimmt keine Anstalten, den Pod auf einem der anderen Nodes zu starten, obwohl dort Replicas des Longhorn-Volumes liegen. Die Anwendung fällt aus, obwohl Sie alles so schön redundant geplant haben. Schuld ist eine gut in den Einstellungen (Setting/General) versteckte Option mit dem Namen „Pod Deletion Policy When Node is Down“. Legen Sie den Schalter um auf „delete-both-statefulset-and-deployment-pod“ und speichern Sie mithilfe des Speichern-Buttons am Ende der Seite. Mit dieser Anweisung wird ein Pod verlagert, wenn ein Node nicht mehr auffindbar ist.

Alternativen

Longhorn ist längst nicht der einzige Storage-Anbieter, der an das CSI von Kubernetes andockt. Die offizielle Liste der CSI-Maintainer enthält über 100 Einträge. Gespeichert werden kann nicht nur auf Festplatten, die sich im Cluster selbst befinden – per CSI lässt sich auf fast alles zugreifen, was in Unternehmen, Kleinbüros oder bei Cloudprovidern steht und Daten aufbewahrt. NAS-Hersteller Synology stellt ebenso einen CSI-Adapter bereit (csi.san.synology.com) wie Amazon AWS oder Microsoft Azure. Es gibt Adapter für Hard- und Software von Dell, HPE und NetApp. Das Prinzip funktioniert immer gleich und wie bei Longhorn: Man folgt der Anweisung des Anbieters für die Installation (meist per Helm-Chart), legt dann eine StorageClass mit individuellen Einstellungen an und hängt die an PersistentVolume-Claims. (jam) c't

Bild: KI Midjourney, Bearbeitung c't

Kubernetes-YAML mit Helm verpacken

Kubernetes-Administratoren definieren ihre Containerumgebungen in YAML – viel YAML. Damit die Clusterverwaltung nicht im Chaos versinkt, sollten Sie sich mit einem Paketmanager wie Helm vertraut machen und eigene parametrisierte Pakete schnüren. Die kann man auch an Kunden und Kollegen weitergeben.

Von **Jan Mahn**

Für Docker- und Podman-Umsteiger fühlen sich die ersten Schritte mit Kubernetes seltsam an: Selbst für einfachste Aufgaben braucht man zig Kubernetes-Objekte. Ein Pod hier, ein Service da, vielleicht noch ein PersistentVolumeClaim und eine ConfigMap dort und dazu eine IngressRoute. Schnell häuft man Hunderte Zeilen YAML an – der Preis für die Flexibilität, jede erdenkliche Feinheit einstellen zu können, wenn man das mal muss.

Entsprechend großzügig mit YAML sind wir auch bei der Cluster-Einrichtung auf unserem Kubernetes-Lernpfad umgegangen: Legen Sie folgende YAML-Schnipsel an und bringen Sie diese per Kubectl in den Cluster, löschen Sie dafür ein altes Objekt, ersetzen Sie ein anderes. Wenn Sie der Anleitung gefolgt sind, haben sich auf Ihrer Festplatte Dutzende solcher Schnipsel angesammelt. Viele der darin definierten Objekte laufen im Cluster, einige haben

Sie vielleicht auch wieder gelöscht, andere per Hand geändert. Zum Ausprobieren ist dieses Vorgehen auch kein Problem, im produktiven Betrieb hat dieser imperative, also aus vielen Befehlen zusammengesetzte Stil, aber zahlreiche Tücken. Denn was wirklich im Cluster läuft, entspricht nicht automatisch dem, was in YAML auf Ihrer Festplatte liegt. Wenn Sie eine Datei mit der Definition eines Objekts löschen, heißt das noch nicht, dass das Objekt aus dem Cluster verschwindet – und andersherum. Wenn dann noch mehrere Admin-Kollegen am Cluster tüfteln, ist das YAML-Chaos perfekt. „Configuration Drift" heißt dieses Problem in der Fachwelt.

Das geht besser: Im deklarativen Stil beschreibt man, wie ein System aussehen soll, und überlässt es einem Werkzeug, nötige Änderungen vorzunehmen, also neue Objekte anzulegen und alte zu entfernen. Die gesamte Definition des Wunschzustands steckt in Paketen, die obendrein noch mit Variablen steuerbar sind. Marktführer für diese Aufgabe ist das Open-Source-Werkzeug Helm (helm.sh), ein Projekt unter dem Dach der Cloud Native Computing Foundation (CNCF). Helm bezeichnet sich selbst als Kubernetes-Paketmanager, und der Umstieg von YAML-Salat auf Helm ist denkbar einfach. Es werden nur gewöhnliche Kubernetes-YAML-Dateien paketiert – eine neue Syntax müssen Sie also nicht lernen, nur bestehendes YAML mit Parametern für Helm versehen.

Das erste Paket

Im dritten Teil des Kubernetes-Einstiegs haben wir beschrieben, wie Sie Helm installieren, mit dem Cluster verbinden und fremde Pakete einsetzen (siehe Artikel „Services und Ingress mit Traefik" auf Seite 52) – denn per Helm installiert man auch die meisten umfangreicheren Kubernetes-Anwendungen wie Traefik oder Longhorn. Das Helm-Kommandozeilenprogramm auf dem lokalen Rechner und ein Testcluster sind die Voraussetzung, um das Folgende nachzubauen.

Um eine eigene Anwendung zu verpacken, brauchen Sie zunächst mal einen Ordner für Ihre Helm-Pakete auf dem lokalen Rechner – bevorzugt direkt ein Git-Repository, dann kann man die Änderungen dank Versionierung sauber nachvollziehen. Öffnen Sie ihn am besten in der IDE Ihres Vertrauens, dann wird das Verschieben und Anpassen von Dateien in der Ordnerstruktur komfortabler.

Starten Sie im Ordner eine Kommandozeilensitzung und richten Sie ein neues Helm-Paket namens my-app ein:

```
helm create my-app
```

Helm legt den Ordner my-app an und füllt ihn direkt mit YAML-Vorlagen für eine Beispielanwendung. Die meisten der erzeugten Dateien werden Sie später löschen oder überschreiben, anhand der Struktur erkennen Sie aber bereits, wie ein Helm-Paket grundsätzlich aufgebaut ist. Deshalb lohnt ein Blick auf diese Musterdateien.

Zunächst ist da eine Datei namens values.yaml. Dort definiert man als Paketersteller Variablen für Details, die der Paketnutzer später beeinflussen darf – Docker-Compose-Nutzer erinnert das Konzept an die .env-Datei mit Umgebungsvariablen. Helm kann aber noch mehr als einfache Ersetzungen, denn die Werte werden von einer sehr flexiblen Template-Engine verarbeitet. Wie man solche Variablen anlegt, kann man gut an folgenden Zeilen nachvollziehen (Zeilen 7 bis 11 der values.yaml):

```
image:
  repository: nginx
  pullPolicy: IfNotPresent
  # Image tag
  tag: ""
```

Bei den Variablen handelt es sich um verschachtelte YAML-Objekte. Das ist schon mal praktischer als eine flache .env-Datei – denn in großen Charts (siehe Artikel „Services und Ingress mit Traefik" auf S. 52) kann es auch mal so viel zu steuern geben, dass es sonst unübersichtlich wird. Was in der values.yaml innerhalb des Helm-Pakets steht, ist immer der Standardwert, den man bei der Installation überschreiben darf, also in dem Moment, in dem man das Chart in den Cluster bringt. Dazu muss man als Nutzer eines Pakets nur eine weitere YAML-Datei anlegen, die lediglich die Elemente enthält, die man überschreiben will. Helm führt dann die Werte aus dem Paket und eigene Werte zusammen. Um zum Beispiel nur den Tag zu überschreiben, reicht es, bei der Installation folgenden Schnipsel an Helm zu übergeben:

```
image:
  tag: alpine
```

Genutzt werden die oben definierten Variablen in der Datei templates/deployment.yaml. Der Ordner templates ist der Ort, an dem man beim Paketieren seine YAML-Dateien ablegt. Bei der Installation des Charts läuft die Template-Engine an und sucht nach

dem Muster {{ }}, das einen zu ersetzenden Platzhalter umrahmt. Um einen Wert aus der Datei values.yaml in ein Kubernetes-Objekt einzusetzen, schreibt man zum Beispiel:

```
{{ .Values.image.repository }}
```

Mit .Values greift man auf die Werte aus der values.yaml zu, danach folgt die Position innerhalb der YAML-Struktur. Im Beispiel oben würde Helm diese Angabe durch den Wert nginx ersetzen. Indem Sie Funktionen per | an eine Variable anhängen, können Sie diese manipulieren – Helm nennt das Konzept Pipeline. Ein Beispiel:

```
{{ .Values.example | quote }}
```

Die Anweisung gibt Helm den Auftrag, die Variable example aus values.yaml zu nehmen und an die vordefinierte Funktion quote zu übergeben. Die setzt Anführungszeichen um den Wert und ist nützlich, wenn man sicherstellen muss, dass ein Wert als String interpretiert wird.

Dabei haben sich die Helm-Entwickler die Funktion quote nicht selbst ausgedacht. Sie nutzen vielmehr die Go-Bibliotheken sprig und Go-Templates, in deren Dokumentationen man noch viele weitere Nützlinge findet (siehe ct.de/w889). Das Beispiel-Chart macht intensiven Gebrauch von der Funktion nindent, die sicherstellt, dass ein Wert um n Leerzeichen eingerückt ist. Folgende Zeile bringt die Variable mit vier Leerzeichen in Position:

```
{{ .Values.example | nindent 4 }}
```

Mit Schleifen und Bedingungen

Die Template-Engine kann auch mit Bedingungen in Form von if und else umgehen und Blöcke nur dann rendern, wenn zum Beispiel eine Variable auf true gestellt wurde. Im Beispiel-Chart sehen Sie das Prinzip in der Datei templates/hpa.yaml. Das ganze Objekt wird von der folgenden Zeile eingeleitet:

```
{{- if .Values.autoscaling.enabled }}
```

Das Ende des If-Blocks folgt erst in der letzten Zeile mit dem Befehl {{- end }}. Passend dazu gibt es auch {{- else }}.

Das Minuszeichen vor if, end und else gehört nicht zur eigentlichen Syntax von Bedingungen. Es weist Helm an, beim Ersetzen alle Leerzeichen und Zeilenumbrüche vorab zu entfernen. Schreibt man nur {{end}}, bliebe im gerenderten YAML-Dokument eine leere Zeile zurück (die, in der das end-Statement selbst stand). Im konkreten Fall ist das nur ein kosmetisches Problem, aber in YAML kann es Fälle geben, in denen man das Minuszeichen gezielt einsetzen muss, damit richtig umbrochen wird. Ein Minuszeichen kann auch hinter einem Befehl stehen, dann räumt Helm Leerzeichen und Umbrüche dahinter auf.

Von if machen große Helm-Pakete fleißig Gebrauch und so sind ganze Teile ihrer Pakete optional. Es ist durchaus üblich, dass sich solche Charts über die Jahre immer weiter aufblähen und mehr und mehr exotische Fälle berücksichtigen – solche Extrawünsche muss man dann über eine Variable explizit einschalten.

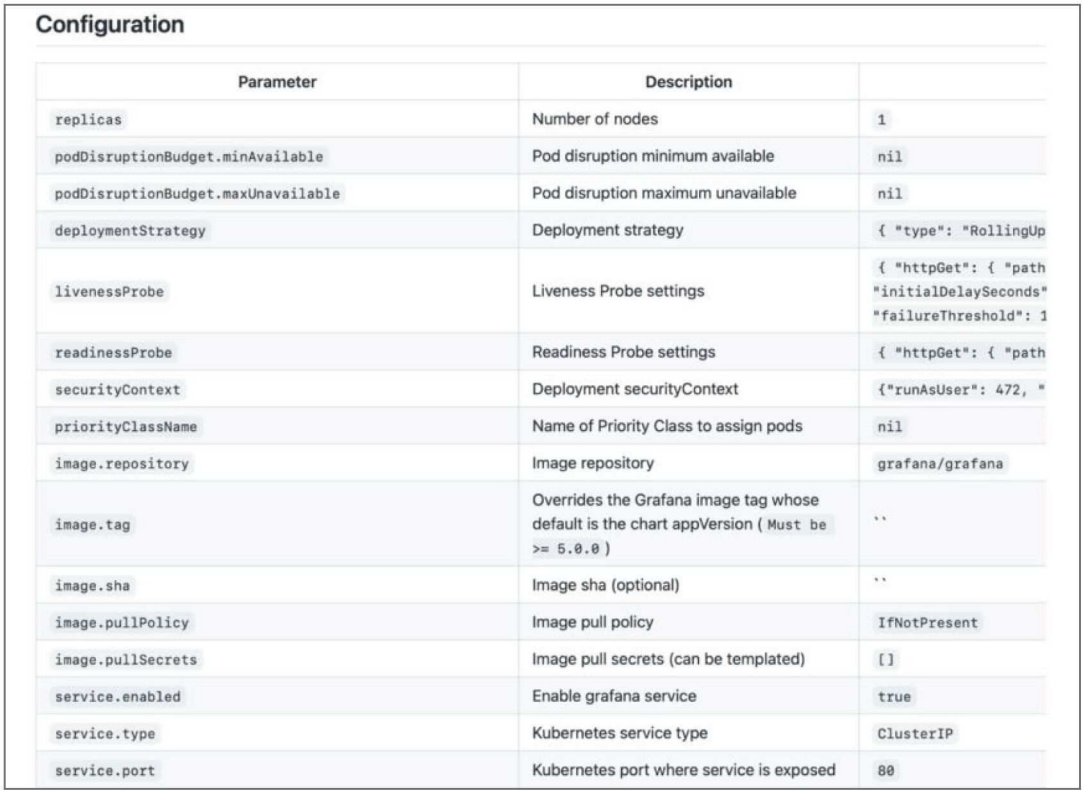
In einer Einführung in eine Programmiersprache folgt nach der Vorstellung von If und Else klassischerweise ein Kapitel über Schleifen – und auch Helms Template-Engine hat da etwas zu bieten. Angenommen, in einer Anwendung sollen mehrere ähnliche Kubernetes-Objekte auf Grundlage einer Liste erstellt werden, dann kann man sich mit einer Schleife viel Kopieren und Einfügen und dadurch redundanten Code ersparen. In der Datei values.yaml definiert man dafür zum Beispiel eine Liste mit Ports:

```
ports:
 - name: ssh
   port: 22
 - name: web
   port: 80
 - name: websecure
   port: 443
```

Aus diesen Werten kann Helm schnell drei Kubernetes-Services machen und die Werte name und port darin einarbeiten:

```
{{- range .Values.ports }}
apiVersion: v1
kind: Service
metadata:
  name: {{ .name }}-service
spec:
  ports:
    - protocol: TCP
      port: {{ .port }}
      targetPort: {{ .port }}
---
{{- end }}
```

Die Liste der Parameter in der Datei values.yaml kann lang und erklärungsbedürftig werden. Das Helm-Chart der Visualisierungssoftware Grafana hat 227 Parameter.



Configuration

Parameter	Description	
replicas	Number of nodes	1
podDisruptionBudget.minAvailable	Pod disruption minimum available	nil
podDisruptionBudget.maxUnavailable	Pod disruption maximum unavailable	nil
deploymentStrategy	Deployment strategy	{ "type": "RollingUp
livenessProbe	Liveness Probe settings	{ "httpGet": { "path "initialDelaySeconds" "failureThreshold": 1
readinessProbe	Readiness Probe settings	{ "httpGet": { "path
securityContext	Deployment securityContext	{"runAsUser": 472, "
priorityClassName	Name of Priority Class to assign pods	nil
image.repository	Image repository	grafana/grafana
image.tag	Overrides the Grafana image tag whose default is the chart appVersion (Must be >= 5.0.0)	``
image.sha	Image sha (optional)	``
image.pullPolicy	Image pull policy	IfNotPresent
image.pullSecrets	Image pull secrets (can be templated)	[]
service.enabled	Enable grafana service	true
service.type	Kubernetes service type	ClusterIP
service.port	Kubernetes port where service is exposed	80

Die Schleife wird mit `range` eingeleitet und mit `end` beendet. In die Schleife mitgenommen werden die Einträge der Liste aus `.Values.ports`, sie sind innerhalb der Schleife mit `.name` und `.port` ansprechbar.

Metadaten

Vollständig ist ein Helm-Chart erst mit der Datei Chart.yaml (in genau dieser Schreibweise). Sie definiert die äußeren Merkmale des Pakets, allen voran Name und Version. Der Befehl `helm create` hat diese Datei bereits angelegt und umfangreich mit Kommentaren versehen. Viel ändern müssen Sie dort nicht. Das Attribut `name` ist bereits mit `my-app` ausgefüllt. Die Beschreibung unter `description` dürfen Sie nach Belieben mit einem beschreibenden Text füllen. Es folgen zwei Versionsangaben, die erklärungsbedürftig sind. Die Angabe `version` soll nicht in Anführungszeichen stehen und muss den Regeln von Semantic Versioning folgen [1], also aus einer Major- Minor- und Patch-Version zusammengesetzt sein (zum Beispiel `1.0.5`). Versioniert werden alle Änderungen am Helm-Chart, also meist an den YAML-Vorlagen.

Unabhängig davon ist der Wert von `appVersion`. Den kann man dafür nutzen, die Version der im Helm-Chart eingebauten Software zu dokumentieren. `version` und `appVersion` müssen keinesfalls identisch sein, schließlich kann man beliebig viele Änderungen am Chart und den Templates vornehmen, ohne eine neuere Version der Software selbst zu nutzen. In der Praxis ist die Angabe von `appVersion` manchmal auch gar nicht sinnvoll nutzbar, zum Beispiel, wenn man mehrere Anwendungen in einem Chart installiert – oder wenn man, wie oben beschrieben, den Tag des Containers in eine Variable schreibt, die der Nutzer setzen darf.

Wenn man nur genau eine Anwendung im Chart ausliefert und sicherstellen will, dass die `appVersion` mit dem ausgelieferten Container-Image überein-

stimmt, kann man auf diesen Wert auch innerhalb eines Templates zugreifen und ihn als Tag für einen Container nutzen. Die Syntax dafür:

```
{{ .Chart.AppVersion }}
```

Schnell erklärt ist eine weitere Datei im Templates-Ordner: NOTES.txt (Schreibweise nur genau so) enthält einen Text, den Helm auf der Kommandozeile anzeigt, wenn man das Paket installiert. Oft reicht hier ein kurzer Verweis auf eine Dokumentation.

Abkürzungen

Das letzte Konzept, das Sie zum Paketieren kennen sollten, steckt in der Datei templates/_helpers.tpl. Darin werden „Named Templates" angelegt. Im Kern sind das Template-Schnipsel, die man mehrfach wiederverwenden und nur einmal schreiben will. In einer Programmiersprache würde man von Funktionen sprechen. Der Befehl `helm create` hat bereits ein paar angelegt, der wichtigste Schnipsel beginnt mit der Zeile:

```
{{- define "my-app.fullname" -}}
```

Den Inhalt dieses Templates muss man nicht zeilenweise verstehen, wichtig ist, was hinten rauskommt: Auf Basis des Namens, der bei der Installation des Charts vergeben wird, generiert Helm für diese Installation einen einmaligen Namen, der maximal 63 Zeichen lang ist und damit den DNS-Spielregeln folgt. Diesen Rückgabewert können Sie benutzen, um Kubernetes-Objekte eindeutig zu benennen. Das ist wichtig, damit Nutzer das Paket auf Wunsch auch mehrfach in einem Cluster einsetzen und unabhängige Instanzen Ihrer Software hochfahren können.

Anwerfen können Sie diesen vorgefertigten Namensgenerator mit:

```
name: {{include "my-app.fullname" .}}
```

Hände in den Code

Genug der Theorie. Mit diesem Wissen können Sie damit beginnen, eine erste Anwendung zu paketieren. Zum Üben können Sie sich YAML schnappen, das Sie oder Kollegen vielleicht schon geschrieben haben. Wenn Sie noch am Anfang Ihrer Kubernetes-Reise sind, schauen Sie in das GitHub-Repository, das wir zu unserer Einstiegsreihe veröffentlicht haben (siehe ct.de/w889). Im Ordner part-4 liegt eine kleine Beispielanwendung mit einer WordPress-Instanz, noch ganz ohne Helm.

Das Überführen von nacktem YAML in Helm ist einfach: Kopieren Sie alle für die Anwendung nötigen Dateien in den Ordner templates des Charts. Über Reihenfolge und Abhängigkeiten von Objekten untereinander müssen Sie sich keinerlei Gedanken machen. Wenn ein Pod zum Beispiel auf einen PersistentVolumeClaim und eine ConfigMap verweist, wird sich das System darum kümmern, dass diese zuerst erzeugt werden.

Im zweiten Schritt können Sie sich ans Parametrisieren machen. Hangeln Sie sich durch alle YAML-Dateien und überlegen Sie, welche Elemente steuerbar sein sollen. Image-Namen und -Tags zum Beispiel sind gute Kandidaten für eine Variable. Für solche Ersetzungen legen Sie eine Variable in der Datei values.yaml an und nutzen sie diese mit der `.Values`-Syntax.

Sobald Sie eine Funktion der Template-Engine eingebaut haben, lohnt es sich, diese mal auszuprobieren. Für eine Trockenübung gibt es den Befehl `helm template`, der ein Chart durch die Template-Engine schickt und das fertige YAML auf der Kommandozeile anzeigt. Navigieren Sie auf der Kommandozeile ins Verzeichnis, in dem der Ordner my-app liegt, und setzen Sie den Befehl ab:

```
helm template ./my-app
```

Die Ausgabe ist nützlich, wenn man prüfen will, ob man mit Ersetzungen, Bedingungen und Schleifen keine groben Fehler eingebaut hat. Um herauszufinden, ob das Paket auch wie geplant läuft, ist ein Testcluster unabdingbar. Zum Probieren sollten Sie selbstredend kein Produktivsystem nutzen und vor jedem Test sorgfältig per Kubectl prüfen, ob Sie im richtigen Kontext arbeiten. Erst wenn Sie sicher sind, dürfen Sie Ihre Anwendung in den Cluster befördern (Kommandozeilensitzung eine Ebene über dem Chart):

```
helm install testapp ./my-app
```

Helm erzeugt eine Helm-Installation mit dem Namen `testapp` aus den Dateien im Ordner. Jetzt können Sie mit bewährten Werkzeugen wie Kubectl und Lens überprüfen, ob die Objekte richtig angelegt wurden und funktionieren. Fehler gefunden? Kein Problem: Einfach Änderungen ins Template schreiben, dann nur noch aktualisieren:

```
helm upgrade testapp ./my-app
```

Der Rest ist klassisches DevOps-Handwerk: schreiben, speichern, testen und im Cluster aktualisieren, bis alles rund läuft.

Um zu testen, wie sich Ihr Chart mit anderen Variablen verhält, legen Sie auf der Ebene oberhalb des Charts eine Datei an, die Sie zum Beispiel custom-values.yaml nennen. Dort tragen Sie nur die Werte ein, die Sie überschreiben wollen. Anschließend speisen Sie diese Werte in die Helm-Installation ein:

```
helm upgrade testapp ↵
↳./my-app --values=custom-values.yaml
```

Genug vom Testen? Der Befehl `helm delete testapp` entfernt Ihre Anwendung spurlos aus dem Cluster. Im GitHub-Repository zum Artikel finden Sie ein fertiges WordPress-Chart, das viele der vorgestellten Konzepte ausnutzt – aber längst nicht alle Funktionen von Helm. In der umfangreichen Dokumentation findet man noch viele Helm-Feinheiten zum Ausprobieren.

Verteilstrategie

Das erste Helm-Chart ist vollgepackt mit Templates. Damit geht es an den nächsten Teil Ihrer Reise durch die Kubernetes-Welt und hier trennen sich die Wege der Reisegruppe. In kleinen Projekten kann man die Charts einfach in ein GitHub- oder GitLab-Repository legen und es bei Bedarf mit Kollegen teilen. Mit den Befehlen `helm install` und `helm upgrade` haben Sie das Handwerkszeug, um die Charts per Hand in Test- und Produktivumgebungen aktuell zu halten. Ausreichend für viele Szenarien und viel besser als YAML-Salat.

Größere Organisationen betreiben für ihre Helm-Charts eine Registry und machen sie darüber Kunden und anderen Nutzern verfügbar. Seit Helm 3.8, das Anfang 2022 erschien, ist diese Aufgabe noch mal deutlich einfacher geworden. Helm-Charts kann man seitdem auch aus einer OCI-kompatiblen Registry installieren.

Unternehmen, die eine solche OCI-Registry selbst hosten (zum Beispiel mit Harbor, siehe ct.de/w889), können ihre Charts dort ablegen. Unter den öffentlichen Registries entwickelt sich GitHubs Angebot ghcr.io gerade zu einem beliebten Ablageort für Helm-Charts. Dort kann man Programmcode, zugehörige Helm-Templates und ein Dockerfile ablegen und die hauseigene CI/CD-Umgebung GitHub Actions ein Image und ein Helm-Chart erzeugen lassen. Wenn Sie mit Actions vertraut sind, schauen Sie sich im Beispiel-Repository mal im Ordner .github/workflows um.

Mit eigenen Helm-Paketen rücken Sie in die Königsklasse der Clusterverwaltung vor. Unser Tipp für die nächsten Schritte: Mit dem Open-Source-Werkzeug Argo CD bekommen Sie eine grafische Oberfläche und ein System, das Helm-Charts automatisch in einem oder auch mehreren Clustern aktuell hält. Die Software kümmert sich stets darum, dass alle definierten Ressourcen an Ort und Stelle sind. Wie das geht, erfahren Sie im nachfolgenden Artikel. (jam) ct

Literatur

[1] Jan Mahn, **Bedeutung 2.0.0**, Warum Versionsnummern nicht willkürlich sind, c't 24/2021, S. 128

Dokumentation und Beispiele

ct.de/w889

Bild: Rudolf A. Blaha

Kubernetes mit Argo CD

Eine containerisierte Umgebung, die automatisch aktuell gehalten wird und so funktioniert, wie man es in einem Git-Repository definiert hat – das ist der Traum von Kubernetes-Admins. Mit Helm und Argo CD kommen Sie diesem Idealzustand entscheidend näher.

Von **Jan Mahn**

Was ist Wahrheit, was Realität? Das ist eine philosophische Frage, die sich zuweilen auch Praktiker wie Kubernetes-Admins stellen müssen. Denn welche Container, Volumes, Services und andere Objekte in einem Kubernetes-Cluster angelegt sind, sieht man dem Cluster von außen nur schwer an. Und die YAML-Definitionen auf den Rechnern der Admins, die per `kubectl apply` in den Cluster geschoben wurden, können höchstens ein trügerisches Gefühl von Wahrheit liefern.

Besser wird die Lage, wenn man, statt YAML-Schnipsel auf mehreren Entwicklermaschinen zu horten, alle Anwendungen in Helm-Charts verpackt. Helm enthält eine Template-Engine, die Kubernetes-

YAML rendert und dabei eine Datei mit Variablen für eine konkrete Umgebung berücksichtigt. Wenn sich alle Admins eines Clusters verabreden und Ressourcen ausschließlich über zentral gelagerte Helm-Charts anlegen, bearbeiten und löschen, ist schon mal viel gewonnen. Eine Einführung in Helm lesen Sie im Artikel „Kubernetes-YAML mit Helm verpacken“ ab Seite 84.

Perfekt ist die Kubernetes-Admin-Welt damit aber noch nicht. Denn wer garantiert schon, dass immer die aktuelle Version der Helm-Charts installiert ist? Wer führt den Befehl `helm upgrade` aus, wenn es Änderungen gibt? Und wo liegen die Dateien mit den individuellen Einstellungen für eine Umgebung, die Helm übergeben wurden? Es fehlt eine gemeinsame Quelle für die eine Wahrheit und ein technisches System, das diese zuverlässig im Cluster durchsetzt.

Auftritt Argo CD

Ein solches System ist Argo CD; eine Software, die – wie viele nützliche Werkzeuge aus dem Kubernetes-Ökosystem – als Open-Source-Software entwickelt wird und die ein offizielles Projekt der Cloud Native Computing Foundation (CNCF) ist. Das CD im Namen steht für „Continuous Delivery“ und beschreibt recht gut, was den Kern der Sache ausmacht, nämlich Änderungswünsche kontinuierlich auszuliefern. Benutzen kann man Argo CD auf vielen verschiedenen Wegen und dieser Artikel zeigt nur eine mögliche Herangehensweise: Argo CD als GitOps-Werkzeug. Der Begriff GitOps (zusammengesetzt aus der Versionsverwaltung Git und Operations) beschreibt einen Administrationsstil, bei dem die gemeinsame Wahrheit, also die Definition der Wunschumgebung, sauber nachverfolgbar in einem Git-Repository liegt. Im folgenden Beispiel arbeitet Argo CD im Zusammenspiel mit Helm. Sie können die Software aber auch mit den Rendering-Systemen Kustomize (kustomize.io) und Jsonnet (jsonnet.org) oder mit nacktem Kubernetes-YAML einsetzen.

Fürs Verständnis hilft zunächst ein Blick aufs große Ganze, bevor es an die Umsetzung im Detail geht. Sie brauchen eine Instanz von Argo CD, die selbst in einem Kubernetes-Cluster läuft. Drei Wege kommen dafür grundsätzlich infrage, die stark mit der Rahmenbedingungen Ihrer Kubernetes-Umgebung zusammenhängen. Einmal kann Argo CD in dem Cluster laufen, in dem auch die zu verwaltenden Anwendungen ausgeführt werden sollen. Das ist eine Strategie für kleinere Umgebungen, vor allem dann, wenn man insgesamt nur einen Cluster zu verwalten hat. Sobald man plant, mit einem Admin-Team mehrere Cluster zu versorgen, bietet es sich an, die Argo-Instanz in ein eigenes kleines Cluster zu stecken – es muss nicht zwangsläufig ein Cluster aus mehreren Maschinen sein, Kubernetes erlaubt auch Ein-Server-Cluster und eine kleine virtuelle Maschine reicht für die Ansprüche von Argo CD aus. Mit einer solchen zentralen Argo-Instanz könnte zum Beispiel eine Entwicklerfirma die Instanzen ihrer Software in den Clustern von vielen Kunden überwachen und aktuell halten. Die dritte Option ist, eine Argo-Umgebung zu mieten – mehr dazu im Kasten „Argo CD zur Miete“ unten.

Argo CD stellt ein API und eine Weboberfläche bereit. Außerdem gibt es ein Kommandozeilenwerkzeug (CLI), um mit Argo CD zu interagieren und ihm Aufträge zu erteilen. Während die Weboberfläche zum alltäglichen Verwalter für die Clusterverwaltung

Argo CD zur Miete

Argo CD ist Open-Source-Software und darf kostenlos im eigenen Cluster installiert werden. Wenn Sie sich diesen Schritt sparen wollen – entweder für einen schnellen Test oder auch in einer Produktivumgebung, können Sie auch eine gemietete Argo-CD-Instanz einsetzen. Solche Instanzen bietet die Firma Akuity an, die den größten Teil zur Argo-Entwicklung beigetragen hat und ihr Geld mit Dienstleistungen rund um Argo verdient. 14 Tage können Sie den Funktionsumfang der Akuity-Plattform kostenlos testen. Wenn Sie sich mit einem GitHub- oder Google-Account einloggen, haben Sie in wenigen Minuten eine fertige Umgebung. Die Testversion finden Sie über die Adresse akuity.io/pricing.

Nach der Testphase müssen sich Firmen, die dauerhaft eine gemietete Argo-Instanz nutzen wollen, an den Vertrieb wenden und einen Preis aushandeln.

werden kann, sollte man von manuellen CLI-Aufrufen eher die Finger lassen und sich keinen imperativen Stil angewöhnen – Argo CD kann man nämlich wundervoll deklarativ bedienen. Dafür legt man Kubernetes-Objekte in dem Cluster an, in dem Argo läuft. Argo selbst läuft dann komplett „stateless“, nutzt also keine persistenten Volumes. Alle Konfigurationen holt es sich aus dem Kubernetes-API. Das macht es einfach, eine Argo-Instanz neu einzurichten oder auf einen neuen Server umzuziehen. Einfach die Kubernetes-Objekte anlegen, dann läuft ArgoCD wieder wie gewünscht. Dieser Artikel lässt die CLI daher links liegen und beschreibt das deklarative Vorgehen.

Damit Argo CD zum GitOps-Werkzeug wird, bekommt es Zugangsdaten für Repositories, die bei einem Git-Hoster wie GitHub oder GitLab (oder in einer selbstgehosteten Instanz) liegen. Es findet darin Anweisungen (in Form von Kubernetes-Objekten), welche Helm-Charts mit welchen Variablen in welche Cluster installiert werden sollen. Einmal eingerichtet, muss man künftig Änderungen nur noch in das Konfigurations-Repository schreiben und kann dann in der Weboberfläche zuschauen, wie Argo CD daraus Objekte erzeugt – und dort kann man auch eingreifen, wenn mal etwas schiefgeht, ein Pod zum Beispiel mal einen manuellen Neustart braucht.

Losgelegt

Bevor Sie sich auf den Weg machen, ein Wort der Warnung: GitOps mit Argo CD gehört zu den anspruchsvollsten Kubernetes-Aufgaben und ist kein Feierabendprojekt. Auf den folgenden Seiten finden Sie zwar eine detaillierte Schritt-für-Schritt-Anleitung, je nach Umgebung werden Sie aber vom beschriebenen Pfad abweichen wollen und in der Dokumentation stöbern müssen. Solide Kubernetes-Kenntnisse sind also dringend nötig. Alle Befehle und YAML-Fragmente, die in diesem Artikel vorkommen, finden Sie in einem GitHub-Repository (siehe ct.de/w2nr) im Ordner /argocd.

Um die Einführung nicht in die Länge zu ziehen, geht diese vom einfacheren Fall aus, in dem Argo CD in dem Cluster läuft, das auch die Anwendungen beherbergt. Dabei stehen Sie vor einem klassischen Henne-Ei-Problem. Argo CD kann zwar andere Anwendungen installieren, aber unmöglich sich selbst aus dem Nichts einrichten. Argo CD bleibt also die letzte Anwendung, die Sie zu Beginn der Reise noch selbst installieren müssen.

Das gelingt schnell über Helm auf der Kommandozeile: Fügen Sie das Argo-Repository hinzu und installieren Sie Argo CD anschließend in den Namespace `argocd`:

```
helm repo add argo ↵
↳https://argoproj.github.io/argo-helm
helm install my-argo argo/argo-cd↵
↳ -n argocd
```

Sie sehen: nichts. Denn die Argo-Weboberfläche und das API haben noch keine Verbindung zur Außenwelt. Wieder so ein Henne-Ei-Problem, denn in einem fertigen Cluster würden Sie fix eine Ingress-Route für Ihren HTTP-Router (wie Traefik oder Nginx) anlegen und den Service `my-argo-argocd-server` inklusive TLS unter einer Adresse im Internet veröffentlichen (siehe Artikel „Sicherheit im Cluster“ ab Seite 68.

Noch fehlt aber der HTTP-Router, den soll Argo ja später installieren. Die Abhilfe für den Moment: Leiten Sie die Weboberfläche temporär per Kubernetes-PortForward auf einen Port Ihres lokalen Rechners weiter:

```
kubectl port-forward ↵
↳service/my-argo-argocd-server↵
↳ -n argocd 8080:443
```

Unter der Adresse localhost:8080 meldet sich jetzt (mit einem zu vernachlässigenden TLS-Zertifikatsfehler) die grafische Oberfläche. Anmelden können Sie sich mit dem Nutzernamen `admin` und einem bei der Installation ausgewürfelten Kennwort, das als Secret im Cluster liegt. Es heißt `argocd-initial-admin-secret`; bequem auslesen können Sie es zum Beispiel mit einem grafischen Werkzeug wie Lens (zu finden über ct.de/w2nr) oder über die Kommandozeile:

```
kubectl -n argocd get secret ↵
↳argocd-initial-admin-secret ↵
↳-o jsonpath="{.data.password}" |↵
↳ base64 -d
```

Kopieren Sie das Passwort und melden Sie sich schon einmal an – viel gibt es auf der Oberfläche aber noch nicht zu sehen.

Die erste Application

Es ist Zeit für Ihre erste Anwendung, die Argo CD installieren und aktuell halten soll. Herzstück der

Konfiguration ist ein Kubernetes-Objekt vom Typ `Application`. Das ist kein Standard-Kubernetes-Objekt, sondern eines, das sich die Argo-Entwickler ausgedacht haben. Kubernetes kennt dafür das Konzept der CustomResourceDefinition (CRD) – eine Anwendung kann solche CRDs registrieren und damit das Kubernetes-API um eigene Objekte erweitern. Eine einfache Argo-CD-Application sieht so aus:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: nginx-application
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/jamct/↵
    ↳kubernetes-einstieg
    targetRevision: main
    path: part-1
  destination:
    name: in-cluster
    namespace: default
```

Die Anwendung heißt `nginx-application` und liegt im Namespace `argocd`. Letzteres ist wichtig, dort erwartet Argo die Definitionen. Mit der `repoURL` verweist die Application auf ein öffentlich erreichbares Git-Repository, das in diesem Fall bei GitHub liegt. Mit `path` und `targetRevision` weist die Konfiguration Argo CD an, im Pfad `part-1` im Branch `main` zu suchen und die Objekte, die darin definiert sind, zu installieren.

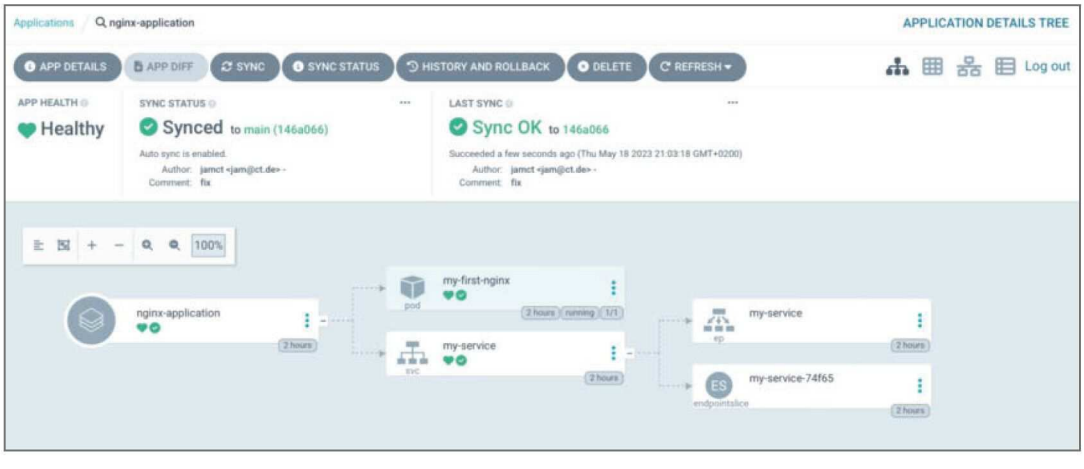
Der Paketmanager Helm kommt in diesem simplen Beispiel noch nicht zum Einsatz, zunächst sollen nur nackte YAML-Dateien ausgeliefert werden. Das kleine Beispiel stammt aus dem Material zu Teil 1 unserer Kubernetes-Einstiegsreihe (siehe Artikel „Der Lernpfad zum Kubernetes-Kenner“ ab S. 38) und enthält einen Nginx-Webserver sowie einen Service, der die Nginx-Seite auf Port 30000 veröffentlicht.

Der letzte Abschnitt `destination` erklärt Argo CD, in welchen Cluster die Anwendung installiert werden soll. Solange man nur einen Cluster hat, schreibt man hier `name: in-cluster`, was „im selben Cluster wie Argo CD“ bedeutet. Wenn Sie das Beispiel nachvollziehen möchten, installieren Sie diese YAML-Definition (zum Kopieren unter ct.de/w2nr zu finden) per `kubectl apply` im Cluster. Öffnen Sie am besten parallel die Weboberfläche, um der Software bei der Arbeit zuzuschauen.

Arbeiten mit der Oberfläche

Wenige Sekunden, nachdem das Objekt im Cluster angekommen ist, zeigt die Weboberfläche die Argo-Application namens `nginx-application` an. Mit einem Klick darauf sehen Sie Details. Argo zeigt Anwendungen in einer Baumstruktur und macht so sichtbar, wie Objekte voneinander abhängen – ein Pod kann zum Beispiel Teil eines Deployments oder eines StatefulSets sein und entspringt dann auch in dieser Ansicht aus seinem Elternelement.

Das größte Problem: Ihre neue Anwendung wird mit einem gelben Gespenster-Symbol als „Missing“



Alles grün: Die Weboberfläche stellt eine Baumstruktur der installierten Objekte dar. Administratoren sehen auf den ersten Blick, ob alles wie geplant funktioniert – anhand von Symbolen in Ampelfarben.

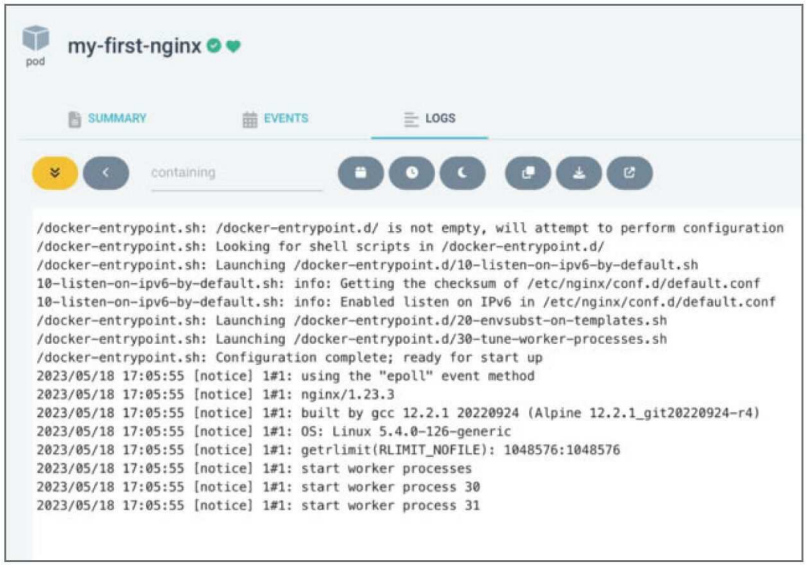
markiert. Ändern können Sie das per Hand mit Klick auf „Sync“ in der Leiste oben. Klicken Sie dann im aufgeklappten Dialog oben rechts auf „Synchronize“. Dann erst startet Argo CD mit der Arbeit und legt Objekte an. Hat das geklappt, steht „Sync OK“ in der oberen Leiste. Weil der Beispiel-Pod my-first-nginx keine LivenessProbe hat, bekommt er sofort ein grünes Herzsymbol, das Zeichen, dass er hochgefahren ist. Bei Pods mit definierter LivenessProbe dauert das ein paar Sekunden.

Der manuelle Klick auf den Sync-Button muss nicht sein. Schließlich wollen Sie ihn nicht bei jeder Änderung an der Definition im Repository betätigen, um die Anwendung zu synchronisieren. Passen Sie stattdessen die sogenannte SyncPolicy in der Definition der Application an. Der folgende Schnipsel gehört ans Ende der oben definierten Application auf die Ebene unterhalb von `spec`, also um zwei Leerzeichen eingerückt:

```
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

Damit wird das automatische Synchronisieren eingeschaltet – bei jeder Änderung am Quell-Repository wird jetzt umgehend reagiert. Mit der Information `prune` können Sie bestimmen, wie Argo CD damit umgehen soll, dass in der Konfiguration Objekte gelöscht wurden. Ist `prune: true` gesetzt, werden diese im Cluster gelöscht. Um sich vor Flüchtigkeitsfehlern im Produktivsystem zu schützen, können Sie die Funktion auch deaktivieren. Wenn Sie dann ein Objekt im Repository entfernen, zeigt es Argo CD in Gelb an und Sie können es per Hand löschen.

Die Funktion `selfHeal` sollten Sie stets aktivieren. Sie kümmert sich darum, dass Objekte sofort zurückkommen, wenn jemand sie im Cluster löscht. Das stellt sicher, dass Definition und Wahrheit immer zusammenpassen. Dieses Verhalten können Sie mit der Beispielanwendung nachvollziehen: Klicken Sie in der Oberfläche auf die drei grünen Punkte auf dem Pod my-first-nginx und löschen ihn mit „Delete“. Der Dialog fragt zur Sicherheit noch mal nach dem Namen, erst dann dürfen Sie ihn löschen. Doch lange verschwindet er nicht aus dem Cluster, Argo arbeitet kurz, dann kommt der Pod zurück. Dieser Mechanismus funktioniert auch, wenn Sie das Objekt über `kubectl delete` löschen. In der Praxis ist die Selbstheilungsfunktion auch dann nützlich, wenn sich ein Pod mal unrettbar verschluckt hat. Einfach über Argo CD löschen und auf automatisch herbeigeschafften Ersatz warten.



Argo CD kann noch mehr, als nur Konfigurationen aus Git-Repositories auszulesen. Die Weboberfläche zeigt zum Beispiel auch Logs von Pods.

Geschützte Umgebung

Eine Funktion konnten Sie bisher nicht testen: Wie Argo CD reagiert, wenn sich etwas im Quell-Repository ändert. Um das auszuprobieren, sollten Sie ein Repository beim Git-Hoster Ihres Vertrauens anlegen (oder auch in einer selbstgehosteten Instanz von zum Beispiel GitLab). In den meisten Fällen sind solche Konfigurations-Repositories nicht öffentlich, also nur mit Zugangsdaten zu erreichen. Dabei sollten Sie Argo CD nicht das Kennwort mit Vollzugriff auf Ihre Repos geben. Stattdessen brauchen Sie ein Token, das lesenden Zugriff auf Ihre Repos gewährt. Um das Folgende nachzuspielen, richten Sie ein privates Repo ein und erzeugen ein solches Token. In der GitHub-Oberfläche finden Sie den Dialog zum Einrichten von Token zum Beispiel unter github.com/settings/personal-access-tokens/new. Die Dokumentation für GitLab finden Sie über ct.de/w2nr.

Jedes private Repository, aus dem Sie Konfigurationen beziehen wollen, muss als Kubernetes-Secret mit Zugangsdaten angelegt werden. Sobald die Um-

gebung größer wird, wollen Sie Ihre Konfigurationen voraussichtlich aus mehreren Repositories mit denselben Zugangsdaten beziehen. Daher lohnt es, direkt sogenannte `repo-creds` als Kubernetes-Secret anzulegen. Die enthalten eine URL, die auf Ihren Benutzernamen oder den Namen Ihrer Organisation zeigt, zum Beispiel auf https://github.com/jamct. Sobald ein solches Secret existiert, nutzt Argo CD es für alle Repositories unter diesem Namen. Für jedes Repository müssen Sie dann dennoch ein Kubernetes-Secret in den Cluster legen, allerdings können Sie Benutzername und Token weglassen. Die Vorlage für die beiden Objekte sieht so aus:

```
apiVersion: v1
kind: Secret
metadata:
  name: gh-creds
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type:↵
    ↳ repo-creds
stringData:
  url: https://github.com/<Name>
  type: helm
  password: <Ihr Token>
  username: <Ihr Name>
  ---
apiVersion: v1
kind: Secret
metadata:
  name: gh-repo
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type:↵
    ↳repository
stringData:
  url: https://github.com/<Name>/<Repo>
```

In der Argo-Weboberfläche können Sie unter Settings/Repositories vorab sicherstellen, dass die Verbindung funktioniert. Ihr Repository sollte mit einem grünen Haken und dem Hinweis „Successful“ aufgelistet werden. Sobald das vorbereitet ist, können Sie das hinterlegte Repository in einer Application nutzen – und zwar direkt mit Helm.

Finale mit Helm

Um das Zusammenspiel mit Helm zu testen, brauchen Sie ein Helm-Chart im Repository. Das können Sie entweder mit `helm create` neu anlegen und selbst mit Beispielen füllen. Oder Sie kopieren den Ordner /base-setup aus unserem Repository (siehe ct.de/w2nr) in Ihres. Dabei handelt es sich um ein Helm-Chart, das ein paar Beispielobjekte anlegt. Außerdem macht es Gebrauch von der Helm-Funktion, Abhängigkeiten in Charts zu definieren. Wie das funktioniert und wie Sie es für eigene Charts einsetzen, lesen Sie im Kasten unten.

Mit dem Beispiel-Chart können Sie auch endlich das Problem lösen, dass Ihre Argo-CD-Umgebung aktuell nur über eine Portweiterleitung erreichbar ist. Das Chart enthält den HTTP-Router Traefik als Abhängigkeit und eine IngressRoute, um Argo zu veröffentlichen. Über eine Argo-Application installieren Sie die Inhalte des Ordners:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: base-setup
  namespace: default

spec:
  project: default
  source:
    repoURL: ↵
    ↳https://github.com/<Name>/<Repo>
    path: base-setup
    helm:
      releaseName: base-setup
      values: |
        traefik:
          enabled: true
  destination:
    name: in-cluster
    namespace: default
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

Anders als im ersten Beispiel in diesem Artikel wird Argo mit `helm:` angewiesen, dass es den Inhalt des Ordners als Helm-Chart interpretieren soll. Der `releaseName` ist optional, `values:` schließlich enthält eine verschachtelte YAML-Struktur, die man in der YAML-Syntax mit | einleitet. Die Variablen dort werden direkt an das Helm-Chart weitergereicht und entfalten dort ihre Wirkung. Schauen Sie in der Weboberfläche zu, wie Argo zahlreiche Objekte installiert – die meisten kommen aus dem eingebundenen Traefik-Chart.

Helm mit Abhängigkeiten

Helm versteht sich nicht nur als YAML-Templating-Engine, sondern vor allem als Paketmanager. Zum Funktionsumfang eines solchen gehört nicht nur das Herunterladen und Installieren von Paketen aus einem Repository – er sollte auch Abhängigkeiten von Abhängigkeiten auflösen, also mit Verschachtelungen umgehen können. Diese Funktion können Sie elegant im Zusammenspiel mit Argo CD einsetzen, indem Sie zum Beispiel ein Chart bauen, das alle Pakete enthält, die Sie gewöhnlich in einem Cluster haben wollen: einen Ingress-Controller (wie Traefik), Monitoringwerkzeuge (wie Prometheus und Grafana) und andere Nützlinge. Legen Sie dazu ein Helm-Chart an und öffnen die Datei Chart.yaml. In der YAML-Struktur ergänzen Sie auf oberster Ebene den Punkt `dependencies`. Traefik integrieren Sie zum Beispiel mit folgenden Zeilen:

```
dependencies:
- name: traefik
  version: "~23.0.1"
  repository: "https://traefik.github.io/charts"
  condition: traefik.enabled
```

Ein Chart mit solchen Abhängigkeiten darf auch eigene Inhalte im Ordner /templates enthalten, Sie können also fertige Anwendungen von fremden Entwicklern mit eigenen Objekten zusammen installieren. Um ein Helm-Chart mit Abhängigkeiten ohne Argo CD zu installieren, brauchen Sie den Befehl `helm dependency update ./mein-chart`, ausgeführt in dem Verzeichnis, in dem Ihr Chart liegt. Damit lädt Helm die Abhängigkeiten in seinen Cache, anschließend können Sie wie gewohnt installieren. Wenn Sie ein solch verschachteltes Helm-Chart im Zusammenspiel mit Argo CD einsetzen, brauchen Sie nichts zu unternehmen, Argo wird die Abhängigkeiten selbst herunterladen.

Helm kennt auch eine Möglichkeit, Variablen aus der values.yaml an eingebundene Charts weiterzugeben. Im Beispiel oben wird ein Paket namens `traefik` eingebunden. Um dessen Standardwerte zu überschreiben, bauen Sie in die Datei values.yaml einfach einen Abschnitt `traefik:` ein und definieren dort die Werte, die überschrieben werden sollen.

Über Variablen können Sie auch steuern, ob eine Abhängigkeit geladen werden soll. Im Beispiel oben ist eine `condition` gesetzt. Die Zeile ist optional – Traefik wird nur dann installiert, wenn die Variable `enabled` unterhalb von `traefik:` auf `true` gesetzt ist.

Jetzt können Sie auch die automatischen Änderungen testen. Fügen Sie dazu ein weiteres Objekt zum templates-Ordner im Chart hinzu und beobachten Sie Argo dabei, wie es dieses umgehend anlegt.

Die nächsten Schritte

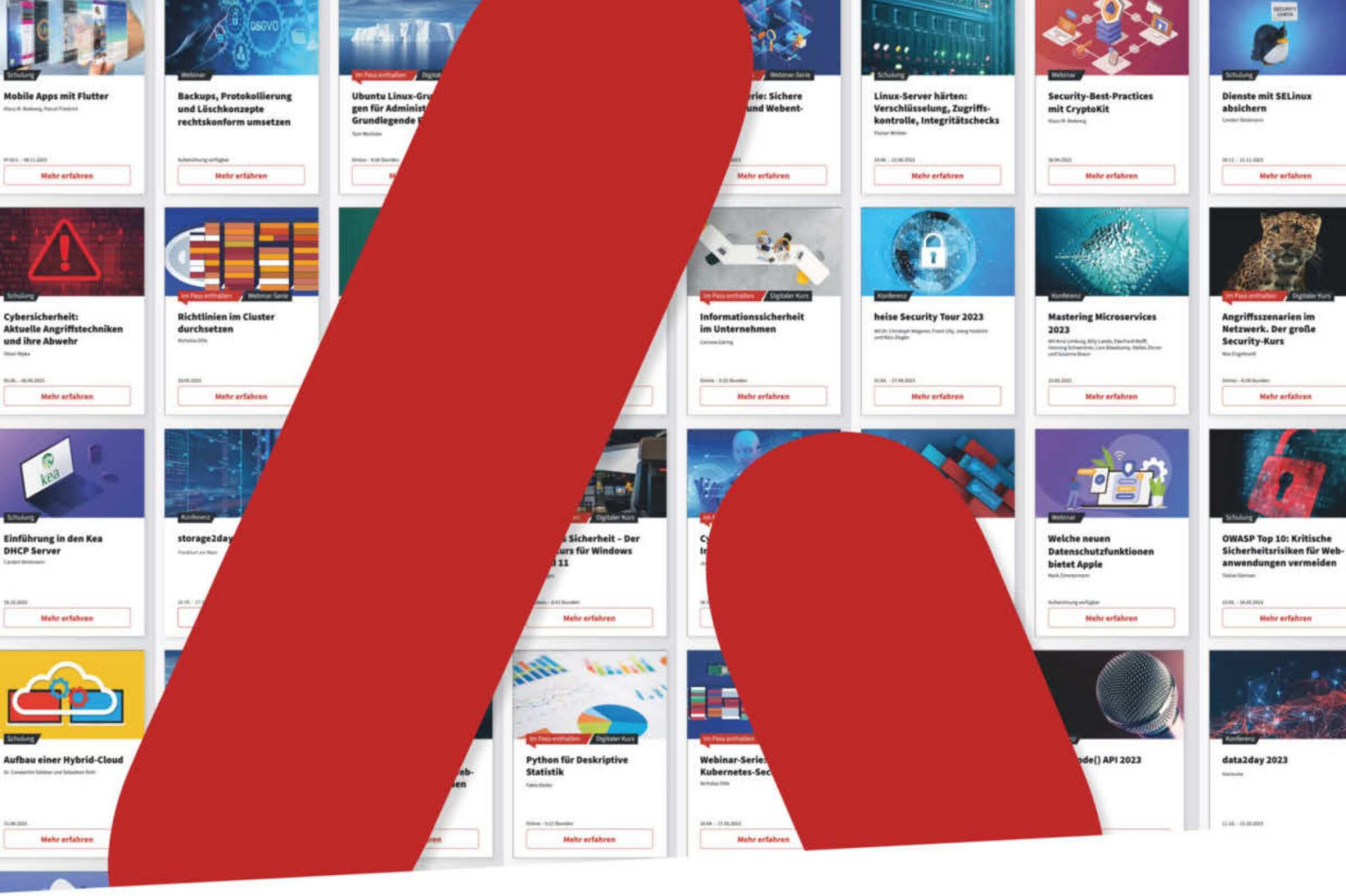
Damit stehen Sie nicht etwa am Ende, sondern vielmehr am Anfang Ihrer Reise durchs GitOps-Universum. Zunächst gibt es noch einen empfehlenswerten Abstraktionsschritt, den Sie einbauen sollten: Im Beispiel oben haben Sie die Application noch selbst per `kubectl apply` installiert. Damit entzieht sie sich der nachverfolgbaren Git-Codeverwaltung und wird spätestens lästig, wenn Sie mehrere Pakete installieren wollen.

Um dieses Problem zu lösen, hat sich unter Argo-Nutzern das Konzept „App of Apps“ etabliert. Dabei legt man eine Application als Meta-Application an, die wie im ersten Beispiel auf ein Verzeichnis in einem Repository zeigt. In diesem Repository liegen die YAML-Schnipsel mit mehreren `Applications`, die zum Beispiel Helm-Charts (aus demselben oder auch einem anderen Repository) nutzen. Per Hand installiert wird nur noch die eine Meta-App, die Argo-Applications erzeugt, die wiederum andere Objekte hervorbringen. Unser Beispiel-Repository zeigt, wie eine „App of Apps“ aussehen kann.

Einen Typ von Kubernetes-Objekten hat dieser Artikel bisher außer Acht gelassen: Geheimnisse wie Zugangsdaten. Diese gehören unter keinen Umständen in Git-Repositories! Um sie zu verwalten und verschlüsselt zu speichern, gibt es Projekte wie Sealed Secrets (sealed-secrets.netlify.app) oder Vault von HashiCorp. Aber das ist eine andere Geschichte. Und auch in Argo CD gibt es noch viel zu entdecken – die Dokumentation erklärt zahlreiche Konzepte, um Feinheiten zu steuern. (jam) c't

Beispiele und Dokumentationen

ct.de/w2nr



Wissenslücken? Nicht mit uns!

Wir helfen Ihnen dabei, die IT-Themen zu lernen,
die heute – und morgen – wichtig sind.

Die Zukunft des Lernens ist digital:
Die heise Academy bietet Ihren IT-Teams die Weiterbildungslösungen an, die Sie benötigen. Lassen Sie Ihre Fachkräfte nach Bedarf und direkt am Arbeitsplatz lernen.

Intensivieren Sie diese Lernerfahrung mit relevanten, topaktuellen Schulungen und Webinaren. Sichern Sie sich das IT-Wissen, das Ihr Unternehmen heute – und morgen – braucht: bei **heise Academy, dem Zuhause Ihrer professionellen IT-Weiterbildung.**

heise Academy

Jetzt entdecken:
heise-academy.de

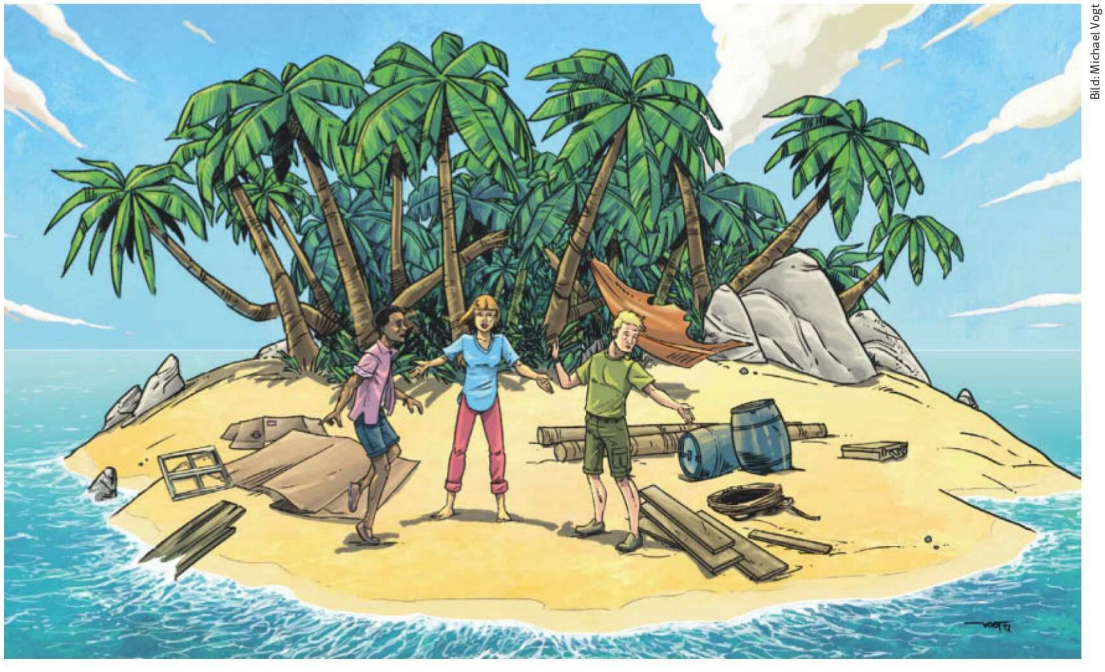


Bild: Michael Vogt

Verteilte Systeme mit Raft-Algorithmus

Wenn Computer gemeinsam als Cluster arbeiten sollen, müssen sie sich zu jeder Zeit auf eine gemeinsame Wahrheit einigen. Der Raft-Consensus-Algorithmus funktioniert mit Mehrheitswahlen, vermeidet so Konflikte und hält unter anderem die Säulen des Internets zusammen: Kubernetes, Docker Swarm, MongoDB und weitere verbreitete Datenbanken.

Von **Jan Mahn**

Wenn Menschen in einer Gruppe zusammenarbeiten sollen, wird es gern mal anstrengend – und es fallen die typischen Sätze, die in jedem Büro zum Alltag gehören: „Das hatte ich doch rumgeschickt!“, „Du hast noch die alte Version, die neue liegt doch im Ordner!“ und „Ach, da warst du im Urlaub, als wir das geändert haben.“ All diese Symptome gehören zu einem großen Problem: Wie stellt man sicher, dass sich alle Mitglieder eines Teams eine gemeinsame Wahrheit, also einen Datenstand als Arbeitsgrundlage teilen?

Dieses Problem ist nicht uns Menschen vorbehalten; auch für Computer ist es keine triviale Aufgabe, reibungslos im Team zu arbeiten und sich auf

eine gemeinsame Wahrheit zu einigen. Dabei haben es Computer untereinander doch vergleichsweise leicht: Sie funktionieren deterministisch, bringen unter identischen Bedingungen also identische Ergebnisse hervor. Auch kennen sie keine Emotionen und menschliche Schwächen wie Neid, Missgunst und krankhaften Ehrgeiz. Einer emotionsfreien Maschine fiele es nicht ein, sich vorzudrängeln oder in fremde Arbeitsbereiche einzumischen.

Es könnte daher so einfach sein, einen Algorithmus zu entwickeln, der eine gemeinsame Wahrheit unter Computern herstellt – und doch hat es Jahrzehnte gedauert, bis sich ein Verfahren durchsetzte, das funktional, robust und leicht zu implementieren ist: der Raft-Konsens-Algorithmus, der heute in vielen verteilten Systemen steckt. Dieser unscheinbare Algorithmus ist zu einer wichtigen Säule des Internets geworden: Die Schlüssel-Werte-Datenbank etcd zum Beispiel arbeitet mit Raft und ist selbst das Fundament für den Container-Orchestrator Kubernetes – und auf dieser Säule wiederum stehen viele große und sehr große Anwendungen. Ohne Raft gäbe es Spotify, Netflix oder Zalando nicht in dieser Form.

Warum ein Konsens-Algorithmus?

Solange ein Computer allein arbeitet, ist alles einfach: Er nimmt Anfragen oder Aufgaben entgegen, beschafft sich die nötigen Daten aus seinem Datenspeicher, nutzt sie für die Bearbeitung und gibt ein Ergebnis zurück. Ein alleinstehender SQL-Datenbankserver zum Beispiel kann lesende und schreibende Anfragen gleichermaßen schnell beantworten und muss sich mit niemandem absprechen. Was er wissen muss, liegt auf seiner Festplatte oder für häufige Fragen im Arbeitsspeicher. Doch ein einzelner Datenbankserver als Rückgrat einer Anwendung hat auch seine Schwächen. Fällt er aus, geht nichts mehr. Und bei höheren Lese- und Schreiblasten bleibt dem Admin nur, immer mehr Hardware in den Einzelgänger zu stecken.

Da wäre es doch schön, wenn man die Daten einem Verbund aus Datenbankservern anvertrauen könnte. Dafür gibt es zwei unterschiedlich komplexe Vorgehensweisen: Im einfachsten Fall ernennt ein Mensch einen Server zum Verantwortlichen (in vielen Anwendungen Master genannt) für alle schreibenden Zugriffe, nur dieser nimmt Schreibaufträge an. Jede Änderung schickt der Master anschließend an weitere replizierende Server, die eine Kopie der Daten pflegen und ausschließlich lesende Zugriffe beantworten oder sogar nur als Reserve für Notfälle bereitstehen. Verteilt man die Lesezugriffe, entlastet das den Master und auch die Stabilität nimmt zu, unter unglücklichen Umständen kann ein Server aber mal veraltete Daten zurückgeben. Außerdem gibt es in einem solchen System eine klare Hierarchie, die ein Mensch erschaffen hat, indem er einen Server zum Anführer und die anderen zu Kopienverwaltern ernannt hat. Hat der Anführer schwerwiegende Probleme, könnte und müsste ein Mensch eingreifen und die Rolle auf einen anderen Server übertragen.

Die Königsdisziplin ist ein Cluster, in dem kein Mensch mehr nötig ist, um Server mit Rollen zu versehen. In einem solchen System sind alle Maschinen gleichberechtigt, und als Anwender kann man sicher sein, dass bei jeder Anfrage die zuletzt gültige Wahrheit in die Bearbeitung einfließt. Eine SQL-Datenbank, die ein solches Konzept umsetzt, ist CockroachDB, die auch in einer quelloffenen Lizenz angeboten wird. CockroachDB nutzt im Kern den Raft-Algorithmus fürs Herstellen einer gemeinsamen Wahrheit und baut darauf all die Funktionen auf, die man von einem SQL-Datenbankserver erwartet.

Dass sich verteilte SQL-Datenbanken eine gemeinsame Wahrheit teilen müssen, leuchtet schnell ein. Einen Konsensalgorithmus brauchen aber nicht nur Systeme, auf denen ein verteilter Datenbestand liegen soll, sondern auch solche, die sich Arbeitsaufträge teilen sollen. Angenommen, Sie möchten eine Umgebung bauen, in der man Rechenaufgaben zur Bearbeitung abwirft – zum Beispiel rechenintensive wissenschaftliche Modelle. Ein großes Heer aus leistungsstarken Maschinen, den Worker-Nodes, soll sich immer eine Aufgabe schnappen und erledigen.

Auch für ein solches System braucht man eine gemeinsame Datenbasis: Sie enthält eine Liste an Aufgaben (sortiert zum Beispiel nach Eingangsdatum oder nach einer Priorität). In dieser Liste muss zuverlässig für jede Aufgabe stehen, ob sie schon ein Worker-Node in Bearbeitung hat und ob sie eventuell schon bearbeitet ist. Nur so ist sichergestellt, dass eine Aufgabe nicht mehrmals bearbeitet wird. Stellen Sie sich das Chaos vor, das in einer Bank entstehen würde, wenn eine Überweisungsaufgabe von mehreren Workern ausgeführt würde, nur weil die Aufgabenliste nicht perfekt synchronisiert wurde.

Was der Algorithmus leisten muss

Ob gemeinsame Datenbank oder System für verteiltes Rechnen, die Anforderungen an ein Verfahren,

Vertrauen und Verrat

Eine Bedingung versucht Raft ausdrücklich nicht zu erfüllen: den Umgang mit byzantinischen Fehlern. So nennt die Forschung in Anlehnung an eine Anekdote mit Generälen aus Byzanz alle Probleme, die durch böswillige Cluster-Mitglieder ausgelöst werden – wenn ein Server im Cluster die anderen anlügt oder Nachrichten mutwillig zurückhält, könnte er die gemeinsame Wahrheit gefährden. Das ist aber in der Praxis kein Problem: Ein Cluster wird für gewöhnlich von einem Administrator aufgesetzt und alle Server werden mit derselben Software ausgestattet.

Eine technische Lösung für byzantinische Umgebungen gibt es abseits von Raft: Blockchains. Auch wenn die Bedingungen in den allermeisten Unternehmensnetzwerken nicht-byzantinisch sind, ist die fixe Idee von Blockchains im Unternehmen bis heute präsent, weil findige Verkäufer erkannt haben, dass Software mit Blockchain interessanter und attraktiver wirkt als Software mit einem vergleichsweise langweiligen Konsens-Algorithmus.

das die gemeinsame Wahrheit koordiniert, sind in beiden Fällen identisch: Gesucht wird ein Ansatz, der zu jeder Zeit sicherstellt, dass eine Anfrage nur mit der zuletzt geschriebenen Wahrheit beantwortet werden kann. Umgehen muss das System mit den Widrigkeiten, die in jedem verteilten System auftreten können. Verzögerungen im Netzwerk, Paketverluste auf dem Weg sowie doppelte oder vertauschte Netzwerkpakete gehören zum Alltag. Die verteilte Datenbank darf also nicht an Schluckauf im Netzwerk scheitern – bestenfalls muss man die einzelnen Server über Rechenzentren verteilen können, die über das Internet miteinander verbunden sind.

Außerdem sollte der Algorithmus sicherstellen, dass temporäre oder dauerhafte Ausfälle einzelner Cluster-Mitglieder nicht dazu führen, dass das Gesamtsystem keine Anfragen mehr bearbeiten kann oder gar falsche Ergebnisse liefert. Neustarts, Updates oder Stromausfälle kommen vor und müssen folgenlos bleiben. Wären solche Ausfälle ein Problem, hätte man lediglich die Komplexität skaliert, in Sachen Hochverfügbarkeit und Fehlertoleranz aber gar nichts gewonnen. Unschön wäre es auch, wenn alle Server bei jeder lesenden Anfrage all ihre Kollegen im Cluster nach der aktuellen Wahrheit fragen müssten – dann hätte man ein ziemlich lahmes System geplant.

Auftritt von Raft

Erfunden und veröffentlicht haben den Raft-Algorithmus Diego Ongaro und John Ousterhout von der Stanford-University im Jahr 2014. Bemerkenswert ist ihre Herangehensweise, die sie in ihrem wissenschaftlichen Paper „In Search of an Understandable Consensus Algorithm“ beschreiben [1]. Schon im Titel taucht die Verständlichkeit als Ziel erstmals auf – und das Thema zieht sich durch den gesamten Artikel. Die zentrale Überzeugung der Autoren bestand darin, dass ein guter Konsens-Algorithmus leicht zu erklären sein muss. Nur dann könne man ihn auch gut und stabil implementieren, verteilte Systeme anständig warten und bei Problemen reagieren. Verständlichkeit soll also Entwicklern und Admins von Raft-Systemen gleichermaßen helfen.

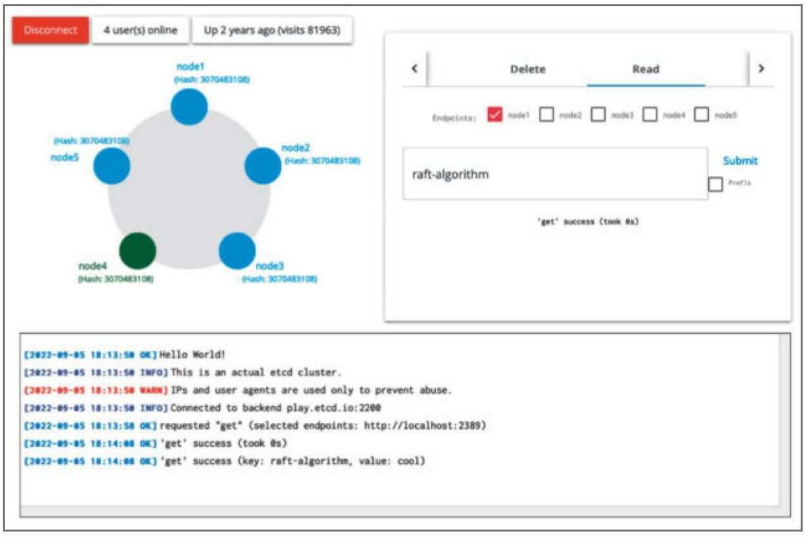
Zu dieser Erkenntnis kamen die beiden Forscher durch schlechte Erfahrungen mit dem damals dominierenden Konsensverfahren namens Paxos oder genauer Multi-Paxos. Das sei, so die Autoren, so vertrackt, dass sie nach zahlreichen Interviews niemanden finden konnten, der Multi-Paxos vollständig und richtig erklären konnte. Die meisten Erklärungsansätze bezögen sich auf den Unterbau Single-Decree-Paxos, der allein schon nicht intuitiv zu verstehen sei – Multi-Paxos legt noch Komplexitätsstufen obendrauf. Kurzum: 2014 war der Stand der Technik ein undurchdringliches Konstrukt, das zu allem Überfluss auch noch immer etwas anders implementiert wurde, gern auch in proprietärer Software ohne öffentlichen Quellcode. Und nicht überall, wo Paxos draufstand, war auch die reine akademische Paxos-Lehre drin.

Anstatt an Paxos herumzudoktern und es zu entschlacken, wie es andere Forscher zuvor versucht hatten, entschieden sich Ongaro und Ousterhout für einen radikalen Neuanfang mit erfrischend wenigen Komponenten. In einem Raft-Cluster kann jeder Server nur einen von drei Zuständen einnehmen: Anführer (Leader), Untertan (Follower) oder Kandidat (Candiate). Im Normalzustand sind diese Rollen klar

aufgeteilt: Es gibt immer nur einen Leader, alle anderen Server sind Follower und es gibt keine Candidates. In diesem Zustand arbeitet ein Cluster die meiste Zeit des Tages und kann Lese- und Schreibaufgaben von Clients entgegennehmen.

Um in diesen Grundzustand zu kommen, ist kein menschliches Eingreifen nötig – beim Einrichten eines Raft-Clusters startet der Administrator einfach mehrere absolut gleichwertige Maschinen und übergibt jeder Maschine eine Liste an Adressen aller anderen Mitspieler. Die wichtigste Bedingung fürs Gelingen: Die Server müssen in einem Netzwerk stecken und einander darüber erreichen, außerdem sollten sie alle dieselbe Uhrzeit haben. Im Original-Paper kommunizieren die Systeme über Remote-Procedure-Calls (RPC), theoretisch könnte man Raft aber auch mit jedem anderen Mechanismus implementieren.

Beim Start erklärt sich jeder Server zunächst zum Follower und wartet auf Anweisungen eines Leaders, genauer auf das Heartbeat-Signal, das nur Leader abgeben – das würde aber ewig dauern, weil ein frischer Cluster anfangs führungslos ist. Damit jetzt nicht alle Server dauerhaft warten, erzeugt jeder Server beim Start eine zufällige Wartezeit (Election Timeout genannt), die innerhalb einer in der Implementierung festgelegten Zeitspanne liegt (die Autoren schlagen zwischen 150 und 300 Millisekunden vor). Jedes Mitglied ist also – per Zufall festgelegt – unterschiedlich geduldig.



Die Key-Value-Datenbank etcd nutzt den Raft-Algorithmus. Unter der Adresse play.etcd.io/play kann man ihre Funktion im Browser erproben.

Derjenige Server, dem zuerst der Geduldsfaden reißt, eröffnet eine neue Wahl, wechselt in die Rolle Candidate und stimmt selbstbewusst schon mal für sich selbst. Dann erhöht er in seinem Speicher die Nummer der Wahlperiode (die Term) und sendet allen anderen Cluster-Mitgliedern eine Wahlbenachrichtigung (mit einem RPC vom Typ `RequestVote`). Einen Wahlkampf gibt es nicht – alle anderen Server, die gerade führungslos vor sich hin warten, nehmen das Angebot dankend an und stimmen umgehend für den ersten Kollegen, der eine Wahl ausgerufen hat, indem sie ihm mit Zustimmung antworten.

Sobald der Candidate die Mehrheit der möglichen Stimmen als Antwort erhalten hat, ernennt er sich zum Leader und beginnt mit seinen Amtsgeschäften: Er versendet Heartbeat-Nachrichten per RPC, alle anderen Server erkennen ihn schlagartig als Leader an und wechseln in den Follower-Zustand. Sollten andere Server durch unglücklichen Zufall parallel eine Wahl angezettelt haben, brechen sie diese bei Erhalt einer Heartbeat-Nachricht sofort ab und folgen ab sofort dem Leader.

Jetzt ist der Cluster bereit, Anfragen von Clients entgegenzunehmen. Beantworten wird solche Anfragen immer nur der Leader, ein unbedarfter Client kennt den aber nicht und beginnt deshalb immer damit, einen zufällig ausgewählten Server zu fragen – der Client braucht also eine Liste aller Server im Cluster. Ist der befragte Server aktuell nicht Leader, muss er die Adresse seines Leaders verraten. Die kann sich der Client erst mal merken und dort künftig nachfragen. Sollte der befragte Server aktuell vom Rest des Clusters getrennt sein, muss der Client sich einen anderen Server zufällig aussuchen und dort anklopfen.

Hat der Client den Leader gefunden, darf er ihn befragen. Noch ist der frische Cluster aber leer, es gibt also nicht viel zu erfahren. Als einfaches Beispiel soll das System eine einzige Zahl verwalten – angenommen, es handelt sich um eine Art Kontostand. In diesem fiktiven System kommt nach erfolgreicher Wahl der erste schreibende Befehl beim Leader an: Die Zahl soll auf 100 geändert werden. In einem Raft-System führt jeder Server zwei Datenstrukturen, einmal ein Log, das alle Schreibbefehle enthält, außerdem die State-Machine, die immer die aktuelle Wahrheit verwaltet.

Zunächst schreibt der Leader den eingegangenen Änderungsbefehl in sein Log, dann schickt er einen

RPC vom Typ `AppendEntries` parallel an alle Follower. Die schreiben die Änderung ebenfalls in ihr Log und bestätigen dem Leader den Empfang. Hat dieser Bestätigungen von mehr als der Hälfte der Server, betrachtet er den Befehl als committed, führt den Änderungswunsch (erhöhe die Zahl auf 100) in seiner State-Machine aus und schickt dem Client die Antwort: Neuer Wert ist 100. Erst dann sagt er allen Followern Bescheid, dass die Änderung committed ist und auch sie die Änderung in ihren State-Machines ausführen können. Mit diesem einfachen Abgleich ist schon einmal sichergestellt, dass die State-Machines aller Server meistens denselben Wert enthalten.

Umgang mit Problemen

Im Kern ist das auch schon das Geheimnis von Raft. In einer perfekten Welt könnte das System für immer so weiterarbeiten – spannend wird es erst, weil die Welt eben nicht perfekt ist und allerhand schiefgehen kann. Das gefühlt schwerste Problem ist in einem Raft-Cluster am schnellsten gelöst: der Ausfall eines Leaders. Angenommen, der Admin schaltet den Server für ein Update aus. Die Follower werden diesen Zustand schnell bemerken, weil die regelmäßigen Heartbeat-Nachrichten ausbleiben. Wie schon beim Einrichten des Clusters wird einem Server zuerst der Geduldsfaden reißen und er wird eine Wahl veranstalten. Dafür macht er sich wieder zum Candidate und eröffnet eine neue Term – ob diese Wahl erfolgreich sein kann, ist abhängig von der Größe des Clusters.

»Durch konsequente Mehrheitswahlen verhindert Raft das bei Clustern gefürchtete Split-Brain-Szenario.«

Ein typischer Raft-Cluster besteht aus drei oder fünf Servern. Wichtig für das Verständnis: Alle Server haben eine Liste mit allen Beteiligten, kennen also auch die Gesamtgröße des Clusters und wissen somit, wo die Grenze für eine Mehrheit bei Wahlen liegt. In einer Umgebung mit drei Servern ist es kein Problem, wenn mal einer ausfällt, selbst wenn es der Leader ist. Einer der anderen stellt sich zur Wahl, der andere stimmt für ihn und die Wahl endet mit zwei von drei Stimmen, also einer Mehrheit.

Zwei Server gleichzeitig dürfen aber nicht ausfallen, dann könnte der verbliebene keine Mehrheit mehr erzielen. Wirkungslos ist es, einen Dreier-Cluster auf vier Mitglieder zu verstärken. Für eine absolute Mehrheit bräuchten die Server in dem Fall drei Stimmen, es darf also wie im Dreier-Cluster nur einer ausfallen. Aus diesem Grund empfehlen alle Dokumentationen von auf Raft basierender Software ungerade Clustergrößen. Bei fünf Mitspielern dürfen schon zwei gleichzeitig ausfallen, die verbliebenen drei können problemlos wählen. Von Clustern mit mehr als sieben Servern raten Raft-Experten eher ab – dann wird das System nur noch träger und nicht mehr stabiler.

Aus einem Zustand ohne Leader kann sich der Cluster in wenigen Millisekunden selbst befreien und mit neuem Leader weiter Clients bedienen. Doch wie sieht es mit dem Ausfall eines Followers aus? Steigt ein solcher aus, fällt das dem Leader spätestens beim nächsten Schreibbefehl auf – und dafür muss man nicht einmal auf einen schreibenden Client warten, denn technisch sind die Heartbeat-Nachrichten auch nur Schreibbefehle (`AppendEntries`) ohne Inhalt.

Der Leader wird also sehr bald versuchen, eine Bestätigung für eine `AppendEntries`-Nachricht zu erhalten. Bleibt diese aus, wird er es immer wieder versuchen – ein anfragender Client muss auf diese Zeremonie aber nicht warten, er bekommt seine Antwort, sobald die Hälfte der Follower geantwortet hat. Ist ein Follower also mal wegen Netzwerkstörungen oder Neustart ein paar Minuten weg, wird er am Ende eine Liste an Änderungsaufträgen bekommen und diese der Reihe nach abarbeiten. Damit es dabei kein Durcheinander gibt, sind alle Schreibaufträge an die Follower nummeriert und enthalten auch immer die Nummer des zuletzt als committed markierten Auftrags sowie die Nummer der Wahlperiode.

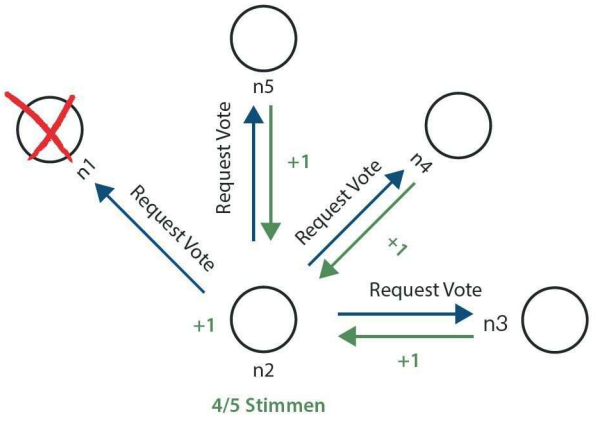
Bevor ein Follower seinem Leader eine Bestätigung schickt, prüft er, ob er die vorangegangene Nachricht erhalten hat. Durch diese Prüfung und die Bestätigung kann jeder Leader eine Liste führen, auf welchem Stand seine Follower sind. Kommt es durch Widrigkeiten des Netzes mal zu einer Abweichung, greift der Leader ein: Er schaut in seiner Liste, bis zu welcher Nachricht der Follower den richtigen Stand kennt, befiehlt dann die Löschung aller späteren Log-Einträge und schickt ihm die Änderungen noch einmal in richtiger Reihenfolge. Sobald er für den letzten Eintrag eine Bestätigung hat, ist der Follower wieder auf Kurs.

Zusätzliche Regeln

Für den Fall von Neuwahlen haben sich die Raft-Autoren einen weiteren Schutzmechanismus ein-

Mehrheitswahl im Raft-Cluster

Bleiben in einem Cluster die Heartbeat-Nachrichten eines Leaders aus, wird ein Server eine neue Wahlperiode ausrufen. Alle anderen Server stimmen für ihn, nachdem sie überprüft haben, ob er die zuletzt gültige Wahrheit kennt.



fallen lassen: Bisher hatte jeder Kandidat, der eine Neuwahl ausgerufen hat, die Stimmen seiner Kollegen sicher. Doch eine Einschränkung (die Leader-Completeness-Property) ist nötig, damit nur ein Server zum Leader wird, der den zuletzt als committed bekannten Stand kennt. Dafür schickt jeder Kandidat im Rahmen der `RequestVote`-Nachricht die fortlaufende Nummer und die Term-Nummer seines letzten Log-Eintrags mit. Die anderen würden ihm die Stimme verwehren, wenn sie selbst einen Eintrag mit höherer Nummer hätten.

Zwei wesentliche Komponenten von Raft, die Wahl des Leaders und auch das Commit von neuen Log-Einträgen, funktionieren nur mit einer Mehrheit. So verhindert Raft einen Zustand, der in anderen Konsens-Algorithmen durchaus möglich und bei Administratoren gefürchtet ist: das Split-Brain-Szenario, bei dem ein Cluster in zwei Teile zerfällt (zum Beispiel durch eine fehlende Netzwerkverbindung) und in dem beide Teile von sich behaupten, die Wahrheit zu verwalten.

Die letzte große Baustelle, mit der sich die Raft-Autoren befasst haben, ist die Mitgliederverwaltung: Unpraktikabel wäre es, einen Cluster bei Änderungen herunterzufahren und mit neuer Mitgliederliste wieder zu starten – Server soll man im laufenden Betrieb hinzufügen und entfernen können. Um das zu gewährleisten, haben sich Ongaro und Ousterhout dazu entschieden, auch die Mitgliederverwaltung wie die Nutzdaten zu behandeln und mit den Raft-Mechanismen zu replizieren. Sobald eine Änderung der Konfiguration (Hinzufügen oder Entfernen eines Mitglieds) verbreitet ist, beginnen alle Follower damit zu arbeiten.

Für neue Server gibt es aber noch eine sinnvolle Ausnahmeregelung: Sie können in den ersten Minuten noch nicht sinnvoll am Cluster mitwirken, weil sie mit einer komplett leeren State-Machine und leerem Log einsteigen und zunächst vom Leader mit alten Logs bombardiert werden. In dieser Phase sind sie weder aktiv noch passiv wahlberechtigt und dürfen in Ruhe die Daten verarbeiten. Sobald sie den letzten Log-Eintrag verarbeitet haben, steigen sie als wahlberechtigtes Vollmitglied ein.

Mit diesen überschaubaren Komponenten erfüllt Raft bereits alle Anforderungen in Bezug auf Stabilität und gemeinsame Wahrheit – ein nicht nur kosmetisches Problem bleibt aber: Die Liste an Log-Einträgen wird länger und länger und verbraucht immer mehr Speicherplatz. Zur Verwaltung einer einzigen Zahl in der State-Machine können nach wochenlangem Betrieb gigabyteweise Logs anfallen, zusätzlich aufgebläht mit Milliarden leerer Heartbeat-Nachrichten.

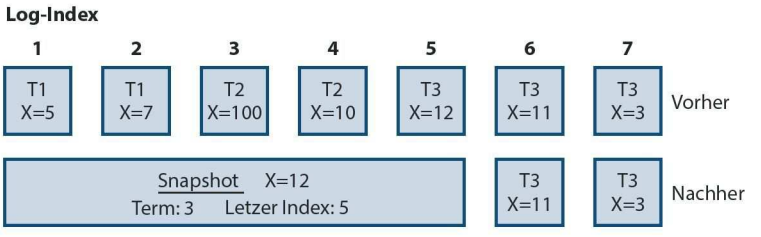
Damit Raft praktikabel bleibt, haben die Autoren Snapshots erfunden, die regelmäßig (abhängig von der Implementierung) angefertigt werden. Ein Snapshot fügt alte Logeinträge zusammen, und anstatt Tausende Änderungswünsche mitzuschleppen, bleibt ein Befehl übrig, der nur das letzte Ergebnis enthält. Die komprimierten Logs werden entsorgt. Die Snapshots entlasten nicht nur die Festplatte, sie machen auch neuen Servern den Einstieg leichter – die bekommen Snapshots vom Leader und sind somit schneller einsatzbereit.

In der Praxis

In freier Wildbahn findet man den Raft-Algorithmus besonders oft in vergleichsweise neuen Anwendungen, die einen Modus für Hochverfügbarkeit mitbringen. Wenn in der Dokumentation einer neuen Datenbank von mindestens drei Servern die Rede ist, ist das ein gutes Indiz für Raft. Neben den schon genannten etcd und CockroachDB sind das häufig Anwendungen aus dem Cloud-Native-Umfeld, also

Log-Komprimierung durch Snapshots

Mit jedem Schreibbefehl wächst das Log und die Datenbank wird unhandlich. Regelmäßig fassen die Server daher alte Logs zu einem Snapshot zusammen, der den letzten Stand enthält – alte Einträge werden entfernt.



solche, die redundant und skalierbar in Containern laufen sollen: Docker Swarm nutzt Raft, ebenso das Nachrichtenverteilsystem NATS (siehe ct.de/wmfp) und die populäre Datenbank MongoDB.

Für Administratoren ist Raft vergleichsweise pflegeleicht. Man startet eine ungerade Anzahl an Servern und überlässt alles Weitere dem Algorithmus – achten sollte man noch darauf, dass sie alle ihre Uhrzeit beim selben NTP-Server beziehen und dass die Latenzen im Netzwerk klein sind. Aussetzer von Maschinen steckt Raft gut weg, theoretisch auch über Stunden und Tage. Es gibt also keine magische Grenze, ab der ein ausgefallener Server nicht wieder hochgefahren werden dürfte – der Leader würde ihm nach kurzem oder langem Ausfall eine Liste mit Aufträgen zuspielen, sodass er sich wieder berappeln kann.

Dennoch sind Admins von auf Raft basierender Software gut beraten, die Abschnitte über „Disaster Recovery“ in der Dokumentation ausführlich zu lesen und die Szenarien einmal in einer vorbereiteten Testumgebung durchzuspielen. Die Anwendungen, die auf Raft aufbauen, führen teils eigene zeitliche Grenzen für ausgeschaltete Server ein und bringen auch Befehle mit, um einen Server sauber für einige Zeit vom Netz zu nehmen (Draining). Und ganz wichtig: Nur weil Raft die Daten auf mehreren Maschinen speichert, heißt das nicht, dass Backups entfallen dürfen. Auch ein Raft-Cluster kann sich mal so verabschieden, dass er sich nicht aus eigener Kraft befreien kann. Regelmäßige Sicherungen sind also Pflicht, und das Einspielen der Backups in einen frischen Cluster sollte man vor dem Einsatz mal ausprobiert haben.

Wenn Sie als Entwickler mit dem Gedanken spielen, Raft in eigene Software einzubauen, die hochverfügbar laufen soll, finden Sie fertig implementierte Raft-Bibliotheken in verschiedenen Programmiersprachen. In vielen Fällen kann man sich das Leben aber einfacher machen und eine auf Raft basierende Datenbank wie etcd oder CockroachDB nutzen. Dann sparen Sie sich zum Beispiel das Implementieren eines Backup-Mechanismus.

Fazit

Einfachheit als Schlüssel zum Erfolg: Die Autoren des Raft-Papers Diego Ongaro und John Ousterhout haben einen für neue Algorithmen ungewöhnlichen Weg gewählt und mit Methoden wie Nutzerumfragen ein Verfahren entwickelt, das primär einfach zu erklären und zu verstehen ist und auf der anderen Seite alle wesentlichen Anforderungen erfüllt. Einen mathematischen Beweis, dass Raft funktioniert, bleibt das Original-Paper schuldig, doch den Beweis hat die Praxis erbracht: Raft machte schnell Karriere, wurde zu einem Quasi-Standard für neu entwickelte verteilte Systeme und beweist seitdem etwa als Teil von Kubernetes jeden Tag, dass es sehr reibungsarm funktioniert. (jam) c't

Literatur

[1] Diego Ongaro und John Ousterhout, **In Search of an Understandable Consensus Algorithm** (Extended Version): raft.github.io/raft.pdf

Raft-Paper
ct.de/wmfp

// heise devSec()

Die Konferenz für sichere Software- und Webentwicklung

11.–13. September 2023 in Karlsruhe

Sichere Software beginnt vor der ersten Zeile Code

Security ist fester Bestandteil der Softwareentwicklung – vom **Entwurf** über den **Entwicklungsprozess** bis zum **Deployment.**

Die **heise devSec** hilft Ihnen dabei mit Vorträgen zu den wichtigsten Themen wie Software Supply Chain, Kryptografie und der Auswirkung von KI auf die Sicherheit.

Aus dem Programm:

// Das ABC sicherer Webanwendungen
// Software Supply Chain Security mit dem SLSA
// Multifaktor-Authentifizierung in der Praxis
// Skalierung von Sicherheit in Kubernetes
// Erweiterung des Secure Development Lifecycle um Privacy by Design
// Wie man mit Mathematik eine Bank übernehmen kann

JETZT **FRÜHBUCHER-TICKETS** SICHERN!

www.heise-devsec.de

Veranstalter: dpunkt.verlag, heise Security, heise Developer

Gold-Sponsoren: opentext™ | Cybersecurity, WIBU SYSTEMS

Bronze-Sponsor: andrena OBJECTS

IMPRESSUM

Redaktion

Postfach 61 04 07, 30604 Hannover
Karl-Wiechert-Allee 10, 30625 Hannover
Telefon: 05 11/53 52-300
Telefax: 05 11/53 52-417
Internet: www.heise.de

Leserbriefe und Fragen zum Heft:
sonderhefte@ct.de

Die E-Mail-Adressen der Redakteure haben die Form xx@ct.de oder xxx@ct.de. Setzen Sie statt „xx“ oder „xxx“ bitte das Redakteurs-Kürzel ein. Die Kürzel finden Sie am Ende der Artikel und hier im Impressum.

Chefredakteur: Jobst-H. Kehrhahn (keh)
(verantwortlich für den Textteil)

Konzeption: Jan Mahn (jam)

Koordination: Pia Ehrhardt (piae), Angela Meyer (anm)

Redaktion: Niklas Dierking (ndi), Jan Mahn (jam), Angela Meyer (anm), Peter Siering (ps)

Mitarbeiter dieser Ausgabe: Markus Richter

Assistenz: Susanne Cölle (suc), Tim Rittmeier (tir), Christopher Tränkmann (cht), Martin Triadan (mat)

DTP-Produktion: Dörte Bluhm, Lara Bögner, Beatrix Dedek, Madlen Grunert, Lisa Hemmerling, Cathrin Kapell, Steffi Martens, Marei Stade, Matthias Timm, Christiane Tümmler, Ninett Wagner

Digitale Produktion: Christine Kreye (Ltg.), Kevin Harte, Thomas Kaltschmidt, Martin Kreft, Pascal Wissner

Fotografie: Andreas Wodrich, Melissa Ramson

Illustration: Rudolf A. Blaha, Frankfurt am Main; Thorsten Hübner, Berlin; Albert Hulm, Berlin; Michael Vogt

Titel: Steffi Martens

Verlag

Heise Medien GmbH & Co. KG
Postfach 61 04 07, 30604 Hannover
Karl-Wiechert-Allee 10, 30625 Hannover
Telefon: 05 11/53 52-0
Telefax: 05 11/53 52-129
Internet: www.heise.de

Herausgeber: Christian Heise, Ansgar Heise, Christian Persson

Geschäftsführer: Ansgar Heise, Beate Gerold

Mitglieder der Geschäftsleitung: Jörg Mühle, Falko Ossmann

Anzeigenleitung: Michael Hanke (-167)
(verantwortlich für den Anzeigenteil),
www.heise.de/mediadaten/ct

Anzeigenverkauf: Verlagsbüro ID GmbH & Co. KG,
Tel.: 05 11/61 65 95-0, www.verlagsbuero-id.de

Leiter Vertrieb und Marketing: André Lux (-299)

Service Sonderdrucke: Julia Conrades (-156)

Druck: Firmengruppe APPL Druck GmbH & Co. KG,
Senefelder Str. 3–11, 86650 Wemding

Vertrieb Einzelverkauf:
DMV DER MEDIENVERTRIEB GmbH & Co. KG
Meßberg 1
20086 Hamburg
Tel.: 040/3019 1800, Fax: 040/3019 145 1815
E-Mail: info@dermedienvertrieb.de
Internet: dermedienvertrieb.de

Einzelpreis: € 22,50; Schweiz CHF 33.90;
Österreich € 24,50; Luxemburg € 25,90

Erstverkaufstag: 20.06.2023

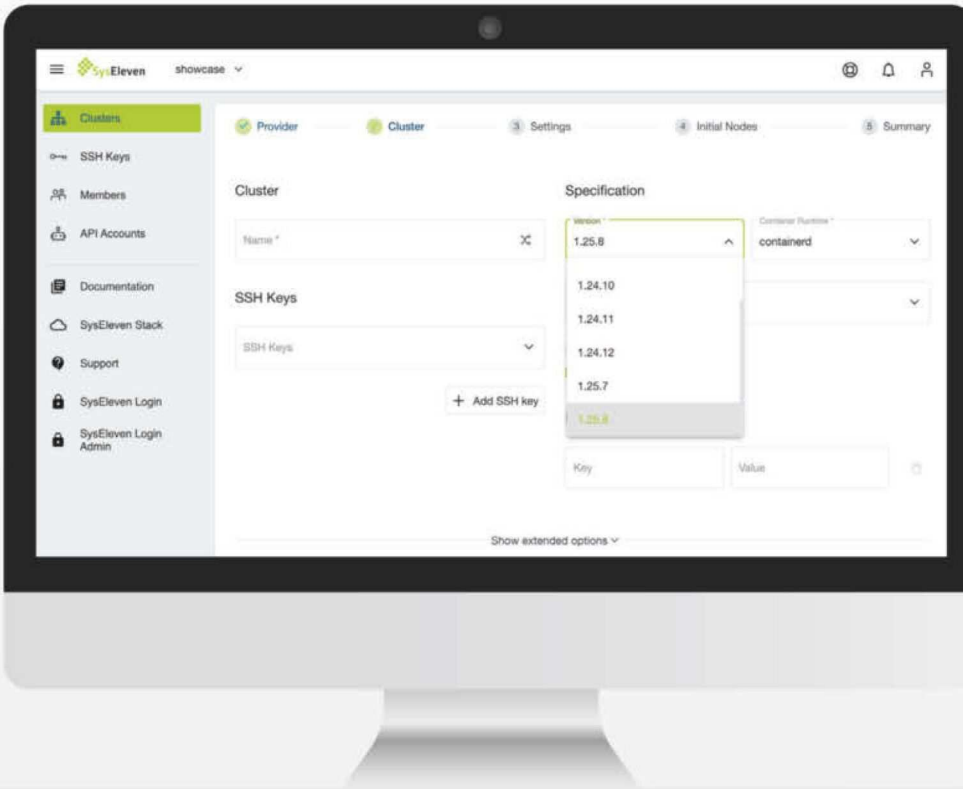
Hergestellt und produziert mit Xpublisher: www.xpublisher.com

Printed in Germany.

// Mastering Cloud. Accelerating Business.

SysEleven

Managed **Kubernetes** – Made in Germany



Jetzt unverbindlich testen

sys11.it/ct

DSGVO-konform
Deine Kubernetes-Cluster laufen auf ISO27001 zertifizierten Rechenzentren in Deutschland.

Building Blocks
Wir stellen kuratierte Dienste wie Observability, Service Mesh, Indexing, Caching, Backup und Recovery bereit.

Developer Tools
Tools wie Terraform, Gitlab und Container Registry sichern eine abteilungsübergreifende Zusammenarbeit.

// Kubernetes zertifizierter Service Provider