# Kubernetes

- Virtuelle Server anlegen und einrichten
- Automatisierte Servereinrichtung
- c't Teil 1: Containerkompetenzoffensive
- c't Teil 2: Dickschiffkapitän
- c't Teil 3: Containervernetzer
- c't Teil 4: Containerladeoffizier
- c't Teil 5: Container-Sicherheitsbegehung
- c't Ergänzung: Speicher

# Virtuelle Server anlegen und einrichten

Nach Anlage der Server müssen sie Konfiguriert werden. Im ersten Schritt werden sie zuerst einmal abgesichert. Dazu wird ein neuer User mit sudo-Rechten angelegt, dem root-User das ssh Login verweigert und das ssh Login grundsätzlich nur noch mit einem Schlüssel erlaubt, die Authentifizierung über user:password also gesperrt.

Natürlich werden alle Ports bis auf das ssh-Port mit Hilfe der ufw gesperrt. Zusätzlich wird file2ban genutzt, um Angriffe auf das ssh-Port zu überwachen und angreifende IPs zu sperren.

```
#
# Angenommener User-Name..: ralf
# Angenommener Domain-Name: die-steffens.eu
#
# !!
# !! Als root auf dem V-Server ausführen
# !!
#
# Swap File anlegen, wenn keines existiert
#
fallocate -l 2G /swapfile
chmod 600 /swapfile
mkswap /swapfile
swapon /swapfile

#
# Neuen Admin-User anlegen
#
useradd -m -U -s /bin/bash -G sudo ralf
passwd ralf

#
# Firewall (ufw) und file2ban installieren
#
apt update
```

```
apt upgrade -y
apt install ufw fail2ban

#
# fail2ban konfigurieren
#
printf "[sshd]\nenabled = true\nbanaction = iptables-multiport\nbantime = 30m" >
/etc/fail2ban/jail.local
systemctl start fail2ban

# !!
# !! Vom lokalen Host den Public Key auf den Server kopieren.
# !! Danach unbedingt testen, ob die Anmeldung ohne Password gelingt!
# !!
ssh-copy-id -i ~/.ssh/id_rsa.pub ralf@die-steffens.eu
ssh ralf@die-steffens.eu

# !!
# !! Als root auf dem V-Server ausführen
# !!
#
# Configure firewall (allow ssh only)
#
ufw allow OpenSSH
ufw enable

#
# Harden ssh
#
sed -i -e '/^\(#\|\)PermitRootLogin/s/^.*$/PermitRootLogin no/' /etc/ssh/sshd_config
sed -i -e '/^\(#\|\)PasswordAuthentication/s/^.*$/PasswordAuthentication no/'
/etc/ssh/sshd_config
sed -i -e '/^\(#\|\)X11Forwarding/s/^.*$/X11Forwarding no/' /etc/ssh/sshd_config
sed -i -e '/^\(#\|\)MaxAuthTries/s/^.*$/MaxAuthTries 2/' /etc/ssh/sshd_config
sed -i -e '/^\(#\|\)AllowTcpForwarding/s/^.*$/AllowTcpForwarding no/' /etc/ssh/sshd_config
sed -i -e '/^\(#\|\)AllowAgentForwarding/s/^.*$/AllowAgentForwarding no/' /etc/ssh/sshd_config
sed -i -e '/^\(#\|\)AuthorizedKeysFile/s/^.*$/AuthorizedKeysFile .ssh\/authorized_keys/'
/etc/ssh/sshd_config
sed -i '$a AllowUsers ralf' /etc/ssh/sshd_config
```

```
#
# Test sshd configuration and restart sshd (if config is ok)
#
sshd -t
systemctl restart sshd

# !!
# !! Vom lokalen Host anmelden
# !!   1. Test muss fehlschlagen - root darf sich nicht mehr anmelden
# !!   2. Test muss gelingen - nur user ralf darf sich (ohne password) anmelden
# !!
ssh root@die-steffens.eu
ssh ralf@die-steffens.eu

# !!
# !! Auf dem V-Server einen Restart ausführen
# !!
sudo shutdown -r now

#
# Wieder einloggen und root werden
#
sudo -s

#
# k3s ports öffnen
#
ufw allow 6443/tcp
ufw allow 2379:2380/tcp
ufw allow from 10.0.0.0/16 to any

#
# k3s installieren
#
mkdir -p /etc/rancher/k3s/
printf "disable: traefik\n" > /etc/rancher/k3s/config.yaml

#
# Auf dem ersten V-Server, der als server das Cluster initialisiert:
#   Das Token wurde mit dem KeyPassXC Passwordgenerator erstellt
```

```
#
curl -sfL https://get.k3s.io |
K3S_TOKEN=rWeygpe4VFENvXA4u7aT7ANodS2q32N53mq9Jnc4xzLR35HRJjqJnSL5YuD4d2jZ sh -s - server --
cluster-init

#
# Für jeden weiteren V-Server, der als k3s server im Cluster arbeiten soll:
#
curl -sfL https://get.k3s.io |
K3S_TOKEN=rWeygpe4VFENvXA4u7aT7ANodS2q32N53mq9Jnc4xzLR35HRJjqJnSL5YuD4d2jZ sh -s - server --
server https://10.0.0.2:6443

#
# Ready
#
```

# Automatisierte Servereinrichtung

Die Servereinrichtung erfolgt in drei Schritten. Die einzelnen Schritte sind notwendig, da bei der Absicherung des Systems der root Account für die Anmeldung gesperrt wird. Es muss also zwischendurch geprüft werden, ob die Installation wie gewünscht durchgelaufen ist. Ansonsten besteht die Gefahr, dass sich niemand mehr auf dem System anmelden kann.

Die einzelnen Skripte liegen lokal vor (*~/Workspace/Kubernetes/vServer*) und sind mit `scp root@<server-ip>:~` auf den neuen Server zu kopieren. Dort sind ihnen mit `chmod +x *.sh` die Ausführungsrechte zuzuweisen. Sie werden in dieser Dokumentation in der Reihenfolge, in der sie ausgeführt werden müssen, dargestellt. Der jeweilige User, der ein Skript ausführen sollte, wird in Klammern angegeben. Alle Skripte benötigen root Rechte, sind also ggf. mit sudo auszuführen.

> Bitte auch die Hinweise vor Skript 3 beachten!

## Skript 1: initServer_1.sh (root)

```
#! /bin/bash
#
# Check if user has root rights
#
idval=`id -u`
if ((0 != $idval))
  then echo "Please run as root (sudo)!"
  exit 1
fi

#
# Swap File anlegen, wenn keines existiert
#
fallocate -l 2G /swapfile
chmod 600 /swapfile
mkswap /swapfile
swapon /swapfile
```

```
#
# file2ban installieren und konfigurieren
#
apt update
apt upgrade -y
apt install fail2ban
printf "[sshd]\nenabled = true\nbanaction = iptables-multiport\nbantime = 30m" >
/etc/fail2ban/jail.local
systemctl start fail2ban

#
# Neuen Admin-User anlegen
#
useradd -m -U -s /bin/bash -G sudo ralf
passwd ralf
```

## Skript 2: initServer_2.sh (ralf)

```
#! /bin/bash
#
# Check if user has root rights
#
idval=`id -u`
if ((0 != $idval))
  then echo "Please run as root (sudo)!"
  exit 1
fi

#
# Harden ssh
#
sed -i -e '/^\(#\|\)PermitRootLogin/s/^.*$/PermitRootLogin no/' /etc/ssh/sshd_config
sed -i -e '/^\(#\|\)PasswordAuthentication/s/^.*$/PasswordAuthentication no/'
/etc/ssh/sshd_config
sed -i -e '/^\(#\|\)X11Forwarding/s/^.*$/X11Forwarding no/' /etc/ssh/sshd_config
sed -i -e '/^\(#\|\)MaxAuthTries/s/^.*$/MaxAuthTries 2/' /etc/ssh/sshd_config
sed -i -e '/^\(#\|\)AllowTcpForwarding/s/^.*$/AllowTcpForwarding no/' /etc/ssh/sshd_config
sed -i -e '/^\(#\|\)AllowAgentForwarding/s/^.*$/AllowAgentForwarding no/' /etc/ssh/sshd_config
sed -i -e '/^\(#\|\)AuthorizedKeysFile/s/^.*$/AuthorizedKeysFile .ssh\/authorized_keys/'
/etc/ssh/sshd_config
```

```
sed -i '$a AllowUsers ralf' /etc/ssh/sshd_config

#
# Test sshd configuration and restart sshd (if config is ok)
#
sshd -t
rc=$?
if [ ${rc} -ne 0 ]; then
  echo "sshd test reports errors - script stopped"
  exit ${rc}
fi
echo "sshd test passed, restart sshd. After restart further root logins are impossible!!"
echo "Please check if all works as expected before logout!"
systemctl restart sshd
exit 0
```

## Skript 3: initServer_3.sh (ralf)

> Dieses Skript benötigt bei Neuaufsetzen des Cluster-Servers (1. Server mit --cluster-init) Anpassungen an den Variablen IP1 und ggf. HOST1 (wenn auch der Servername geändert wurde).

```
#!/bin/bash
#
# Check if user has root rights
#
idval=`id -u`
if ((0 != $idval))
  then echo "Please run as root (sudo)!"
  exit 1
fi

HOST1="kub-1"
IP1="10.0.0.2"

#
# k3s installieren
#
mkdir -p /etc/rancher/k3s/
printf "disable: traefik\n" > /etc/rancher/k3s/config.yaml
```

```
#
# Auf dem ersten V-Server, der als server das Cluster initialisiert:
#   Das Token wurde mit dem KeyPassXC Passwordgenerator erstellt
#
hostval=`hostname`
if [ "$hostval" != "$HOST1" ]; then
    echo "Install next server ($hostval) in cluster ..."
    curl -sfL https://get.k3s.io |
K3S_TOKEN=rWeygpe4VFENvXA4u7aT7ANodS2q32N53mq9Jnc4xzLR35HRJjqJnSL5YuD4d2jZ sh -s - server --
server https://$IP1:6443
  else
    ech  "Install first server for cluster ..."
    curl -sfL https://get.k3s.io |
K3S_TOKEN=rWeygpe4VFENvXA4u7aT7ANodS2q32N53mq9Jnc4xzLR35HRJjqJnSL5YuD4d2jZ sh -s - server --
cluster-init
fi
```

# c't Teil 1: Containerkompetenzoffensive

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 Umstieg: Der erste Teil zeigt, wie Sie aus drei Linux-Servern einen Cluster bauen.

Von Jan Mahn

c't 22/2022, Seite 164

## kompakt

- Kubernetes führt dieselben Container-Abbilder aus wie Docker, mehrere Server können als Cluster zusammenarbeiten.
- Cluster kann man fertig eingerichtet mieten oder auf eigenen virtuellen Maschinen betreiben. Entscheidend ist die Wahl der Kubernetes-Distribution.
- Gesteuert wird Kubernetes per Kommandozeile oder grafischer Oberfläche aus der Ferne.

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 auf Seite 166. 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 in [1].

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.

## 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.

# Eigene Erfahrungen statt Theorie

Dieser Artikel 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. Dieser Artikel ist Teil einer Serie, denn nach einem einzelnen Text sind Sie noch kein Kubernetes-Experte. 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-P
>
> 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.

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. In [2] finden Sie eine Marktübersicht europäischer und US-Anbieter in diesem Geschäft. 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 in [3].

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/yepd).

# 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.

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

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/
```

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>:6
```

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/yepd).

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.

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

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 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/yepd.

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 oben.

# 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 Einrichtung von Servern, Firewalls, DNS-Einträgen und schließlich k3s.

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

# 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/yepd 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. In einer der nächsten c’t-Ausgaben erfahren Sie im nächsten Teil dieser Reihe, warum Container in Kubernetes in sogenannten Pods stecken, wie Sie solche erzeugen und mit der Außenwelt verdrahten. *(jam@ct.de)*

## 1. Literatur


2. Jan Mahn, Zu neuen Ufern, Nach dem Hype: Docker verstehen und loslegen, c’t 24/2021, S. 146
3. Holger Bleich und Jan Mahn, Wolkenangebotsvielfalt, Sechs europäische Cloudprovider im Überblick, c’t 21/2021, S. 62
4. Jan Mahn, Gemeinsame Wahrheit, Wie verteilte Systeme dank Raft-Algorithmus zusammenarbeiten, c’t 21/2022, S. 146


**Dokumentation und Beispiele:**ct.de/yepd

# c't Teil 2: Dickschiffkapitän

Der Weg zum Kubernetes-Kenner muss nicht steinig sein. Kleine Server und solides Docker-Vorwissen reichen für den Einstieg. 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

c't 23/2022, Seite 158

## kompakt

- In Kubernetes stecken Container in sogenannten Pods. Kubernetes kann sehr gut überwachen, wann ein solcher bereit ist.
- Damit Netzwerkverkehr in einem Pod ankommt, braucht man zusätzlich einen Service – mit eingebautem Load Balancer.
- Anders als in Docker können Sie mit etwas Konfigurationsarbeit sicherstellen, dass Updates ohne Auswirkungen auf den Betrieb stattfinden.

Container-Umgebungen funktionieren im Prinzip alle gleich: Im Hintergrund läuft eine Container-Runtime, die Container auf Basis eines Abbildes (Container-Image) startet, den Prozess im Container vom Rest des Betriebssystems trennt und mit Ressourcen wie Netzwerk und Speicherplatz versorgt. Das ist auf einem einzelnen Raspberry Pi mit Docker nicht anders als in einem gigantischen Kubernetes-Cluster, der die Dienste eines Weltkonzerns bereitstellt. Und auch wenn das Prinzip dasselbe ist, gibt es beim Umstieg von Docker auf Kubernetes eine Menge Tücken und Irrwege. Die will diese mehrteilige Reihe vermeiden und Docker-Kennern einen praktikablen Lernpfad aufzeigen. Im ersten Teil haben Sie erfahren, wie aus drei Linux-Maschinen ein Kubernetes-Cluster wird [1]. 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 – all diese Schritte sind in Teil 1 beschrieben.

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/ynfw.

# 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.

Auf keinen Fall sollte man seine gesamte Anwendung in einen einzigen Pod stopfen.

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-Schnipsel 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.

> Jedes Objekt, das man aus einer YAML-Definition in den Cluster bringt, reichert Kubernetes mit
>
> 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.

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

> Wenn die Anforderungen größer werden: Ein Kubernetes-Deployment ist eine Schablone für Pc
>
> Wenn die Anforderungen größer werden: Ein Kubernetes-Deployment ist eine Schablone für Pods – Sie bestimmen, wie viele Kopien für Ihre Anwendung sinnvoll sind.

# 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.

Die Zeiten von Wartungsfenstern sind mit RollingUpdate vorbei.

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. In der Praxis ist das sehr häufig der geeignete Weg und auch die Nginx-Container kann man so prüfen.

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. Der kleine Nginx-Container ist schnell mit einer ReadinessProbe versehen:

```
    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 (der Prozess also nicht mit einer Fehlermeldung und einem Fehlercode aussteigt), 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. Viele fertige Images haben einen solchen Endpunkt ebenfalls eingebaut. 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.

Niemals darf eine LivenessProbe von einem anderen Pod abhängig sein.

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 im Cluster gebaut. Stellen Sie sich vor, Sie haben eine Datenbank im Cluster installiert. Außerdem diverse Pods mit Microservices, also Anwendungen, 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.

# Eine Frage des Stils

Bisher haben Sie Kubernetes ausschließlich mit Kubectl-Aufrufen und lokalen YAML-Dateien bedient. Diese Herangehensweise bezeichnet man als imperativ, weil sie aus einer Reihe von Befehlen besteht. Was am Ende im Cluster läuft, kann man nur erschließen, wenn man alle ausgeführten Befehle in der richtigen Reihenfolge kennt. Und das ist auch das größte Problem am imperativen Administrieren: Besonders in Teams mit mehreren Admins weiß schnell niemand mehr, wer wann welche YAML-Dateien von seinem Computer per Kubectl ins Cluster geschoben hat – vorbei ist es mit der schönen Reproduzierbarkeit und der Cluster wird zur Objekthalde. `kubectl apply` kann immer nur Objekte anlegen oder bearbeiten, nicht aber alte Objekte löschen. Daher ist

das Administrieren per Kubectl noch nicht der Königsweg, sondern nur der Einstieg und ein Mittel für Reparaturen und Experimente. Erstrebenswert ist stattdessen der deklarative Stil. Statt den Cluster mit Befehlen wie „Erzeuge einen neuen Pod“, „Lösche einen alten Pod“ und „Ändere eine Einstellung in einem anderen Pod“ zu versorgen, sagt man „Ich hätte gern den folgenden Zustand.“ Was zum Erreichen dieses Zustands nötig ist, muss und soll kein Mensch entscheiden, der darf sich darauf verlassen können, dass die Maschine sich darum kümmert.

> Mit dem Image containous/whoami sehen Sie, welcher Pod die Anfrage bearbeitet hat.

Mit dem Image containous/whoami sehen Sie, welcher Pod die Anfrage bearbeitet hat.

Das aktuell populärste (aber nicht das einzige) Werkzeug, um das von Haus aus imperative Kubernetes-API deklarativ zu nutzen, heißt Helm (helm.sh). In der Selbstbeschreibung bezeichnen die Entwickler ihr Kommandozeilenprogramm als Paketmanager für Kubernetes. Das greift aber zu kurz, Helm enthält zusätzlich eine Templating-Engine, die Werte aus Konfigurationsdateien in YAML-Gerüste einbaut und macht ganz nebenbei deklaratives Arbeiten möglich.

Im dritten Teil dieser Reihe, der in einer der nächsten Ausgaben erscheint, erfahren Sie, wie Sie fertiges YAML für Helm einpacken, mit Parametern versehen und sich einem deklarativen Stil annähern. Als Beispiel dient eine Anwendung mit mehreren Pods, die miteinander kommunizieren müssen. Wenn Sie bis dahin das Wissen aus diesem Teil anwenden wollen, versuchen Sie doch einmal, eine containerisierte Anwendung, die Sie aktuell mit Docker nutzen, in einen Pod zu verpacken und mit einer LivenessProbe auszustatten. *(jam@ct.de)*

**Dokumentation und Beispiele:** ct.de/ynfw

# c't Teil 3: Containervernetzer

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

c't 25/2022, Seite 162

## kompakt

- In einem Kubernetes-Cluster sprechen sich Dienste untereinander über Services an, die ganz nebenbei Lastenverteilung machen.
- Mit wenigen Kubernetes-Objekten läuft die Blog-Software WordPress mit einer Datenbank im Cluster.
- Um komplexe Software im Cluster zu installieren, steht der Paketmanager Helm bereit.

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 zweiten Teil dieser Reihe [1] 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 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/yp9w.

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 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 auf Seite 164, Frontend und Backend getrennt. Die YAML-Definition haben wir Ihnen über ct.de/yp9w zum Download bereitgestellt. Laden Sie diese Datei herunter und bringen die Definition per Kubectl in Ihren Cluster.

## 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
```

# WordPress-Frontend

```
---
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
```

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 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, die wir schon ausführlich vorgestellt haben [2].

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 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.

> WordPress ist das ideale Beispiel für eine Anwendung, die aus mehreren Komponenten besteh
>
> 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 .

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 [3]. 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 Befehl `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.

> Helm bezeichnet sich selbst als Paketmanager. Mit dem Kommandozeilenprogramm installiere
>
> 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.

# 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/yp9w 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/yp9w) sowie in [3].

```
---
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.

# 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 vierten Teil dieser Reihe, der in einer der nächsten Ausgaben erscheint, soll dieses Problem beseitigt werden – denn wie bei fast allen Aufgaben hat das Kubernetes-Ökosystem dafür ein paar sehr umfangreiche Lösungen zu bieten. (*jam@ct.de*)

## 1. Literatur


2. Jan Mahn, Dickschiffkapitän, Auf dem Lernpfad zum Kubernetes-Kenner, Teil 2, c’t 23/2022, S. 158
3. Jan Mahn, Verteile und herrsche, Verteilte Datenbanken mit CockroachDB, c’t 26/2019, S. 140
4. Jan Mahn, HTTP-Einweiser, Eingehenden HTTP-Verkehr mit Traefik routen, c’t 17/2019, S. 158

**YAML-Dateien und Dokumentation:** ct.de/yp9w

# c't Teil 4: Containerladeoffizier

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

c't 26/2022, Seite 130

## kompakt

- Kubernetes ist darauf ausgelegt, neben Containern auch Konfigurationen und Anwendungsdaten zu speichern – ganz so simpel wie mit Docker gelingt das aber nicht.
- Für Konfigurationsdateien kennt Kubernetes die Objekte ConfigMap und Secret.
- Für Volumes mit persistenten Daten hat Kubernetes Abstraktionsschichten eingebaut. Mit den Objekten StorageClass und PersistentVolumeClaim steuern Sie genau, wo Ihre Daten landen.

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

In den ersten drei Artikeln dieser Reihe haben Sie erfahren, wie Sie einen Cluster einrichten und darauf zugreifen [1], was es mit Containern und Pods auf sich hat [2] und wie Netzwerkverkehr von innen und außen in die Container kommt [3]. Sind diese Voraussetzungen geschaffen, kann es mit Speicherplatz weitergehen.

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/y45k 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-Datenbank abzuspeichern (siehe ct.de/y45k). 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/y45k. 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 sogenannten 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/y45k). 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/y45k.

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.

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 auf Seite 132 sehen Sie, wie ein Datenbank-Container seinen Speicherplatz vom Provisioner `local-path` bekommt.

```
---
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
```

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).

> Die CNCF-Landkarte (landscape.cncf.io) zeigt, wie vielfältig das Angebot an Storage-Anbindung
>
> 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.

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/y45k 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 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.

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 zum Beispiel auch ein Backup-Mechanismus, der die Inhalte von Volumes auf externe Datenhalden (zum Beispiel S3-Speicher) kopieren kann.

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

Grundsätzlich ist Longhorn schnell eingerichtet und der Local-Path-Provisioner ersetzt. Folgendes Problem ist jedoch für das Kubernetes-Umfeld nicht ganz untypisch: Zum Zeitpunkt, als dieser Artikel entsteht, ist Kubernetes 1.25 aktuell. Diese Version haben Sie auch installiert, wenn Sie der Anleitung aus [1] gefolgt sind. Mit 1.25 haben die Kubernetes-Entwickler alte Zöpfe abgeschnitten, sogenannte PodSecurityPolicies, um genau zu sein. Diese kommen während der Longhorn-Installation aber noch vor, auch wenn sie verzichtbar sind. Aktuell arbeiten die Longhorn-Entwickler daran, diese Konstruktion zu entfernen; versprochen ist das Longhorn-Update, das mit dem neuen Kubernetes zurechtkommt, für Ende des Jahres 2022 – einen Einstieg in Longhorn liefern wir Anfang 2023 nach. Für Ihre Kubernetes-Karriere können Sie sich notieren: Es ist nicht immer ratsam, zu den Ersten zu gehören, die eine neue Version ausprobieren.

# 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 (wie Longhorn) 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 links sehen Sie, welche Schritte Sie bereits hinter sich gebracht haben und was es noch zu erkunden gibt. In einer der nächsten

Ausgaben erfahren Sie im fünften Teil dieser Reihe, 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.

Um das bisher gesammelte Wissen zu vertiefen, sollten Sie sich als Docker-Umsteiger Ihre bereits erprobten Docker-Compose-Rezepte schnappen und Erfahrungen sammeln, indem Sie eigene Anwendungen von Docker in Kubernetes übersetzen. (*jam@ct.de*)

## 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 ver- packen
- Logging und Monitoring
- automatische Cluster-Verwaltung und GitOps
- Anwendungen für Kubernetes ent- wickeln

## 1. Literatur


2. Jan Mahn, Containerkompetenzoffensive, Auf dem Lernpfad zum Kubernetes-Kenner, Teil 1, c’t 22/2022, S. 164
3. Jan Mahn, Dickschiffkapitän, Auf dem Lernpfad zum Kubernetes-Kenner, Teil 2, c’t 23/2022, S. 158
4. Jan Mahn, Containervernetzer, Auf dem Lernpfad zum Kubernetes-Kenner, Teil 3, c’t 25/2022, S. 162


**Beispiele und Dokumentationen:** ct.de/y45k

# c't Teil 5: Container-Sicherheitsbegehung

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

c't 3/2023  S. 154

## c't kompakt

- Webseiten will man heute per HTTPS veröffentlichen. Mit Traefik in Kubernetes ist das kein Problem.
- Damit Pods im Cluster nur die Dienste erreichen, die Sie freigeben, gibt es NetworkPolicies.
- Sobald mehrere Nutzer am Cluster arbeiten, sollten Sie Rollen vergeben und sich um Rechteverwaltung kümmern.

Geschafft – in den ersten vier Teilen dieser Reihe haben wir Sie auf dem Weg vom Docker-Nutzer zum Kubernetes-Cluster-Betreiber begleitet und als Beispiel Schritt für Schritt eine WordPress-Instanz zusammengebaut [1]. 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 entscheidendes 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 sogenannte 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/yqsq), 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 IngressRoute 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 Log4Shell-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:
```

```
    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.

> Mit dem NetworkPolicy-Editor von Cilium hat man passende Regeln ohne Tipperei schnell zusa
> Mit dem NetworkPolicy-Editor von Cilium hat man passende Regeln ohne Tipperei schnell zusammengebaut.

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/yqsq) – 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. Der Code sieht folgendermaßen aus:

```
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>
```

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. *(jam@ct.de)*

## 1. Literatur


2. Jan Mahn, Containerladeoffizier, Auf dem Lernpfad zum Kubernetes-Kenner, Teil 4, c’t 26/2022, S. 130


**Beispiele und Dokumentation:** ct.de/yqsq

# c't Ergänzung: Speicher

## Redundanter Container-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.

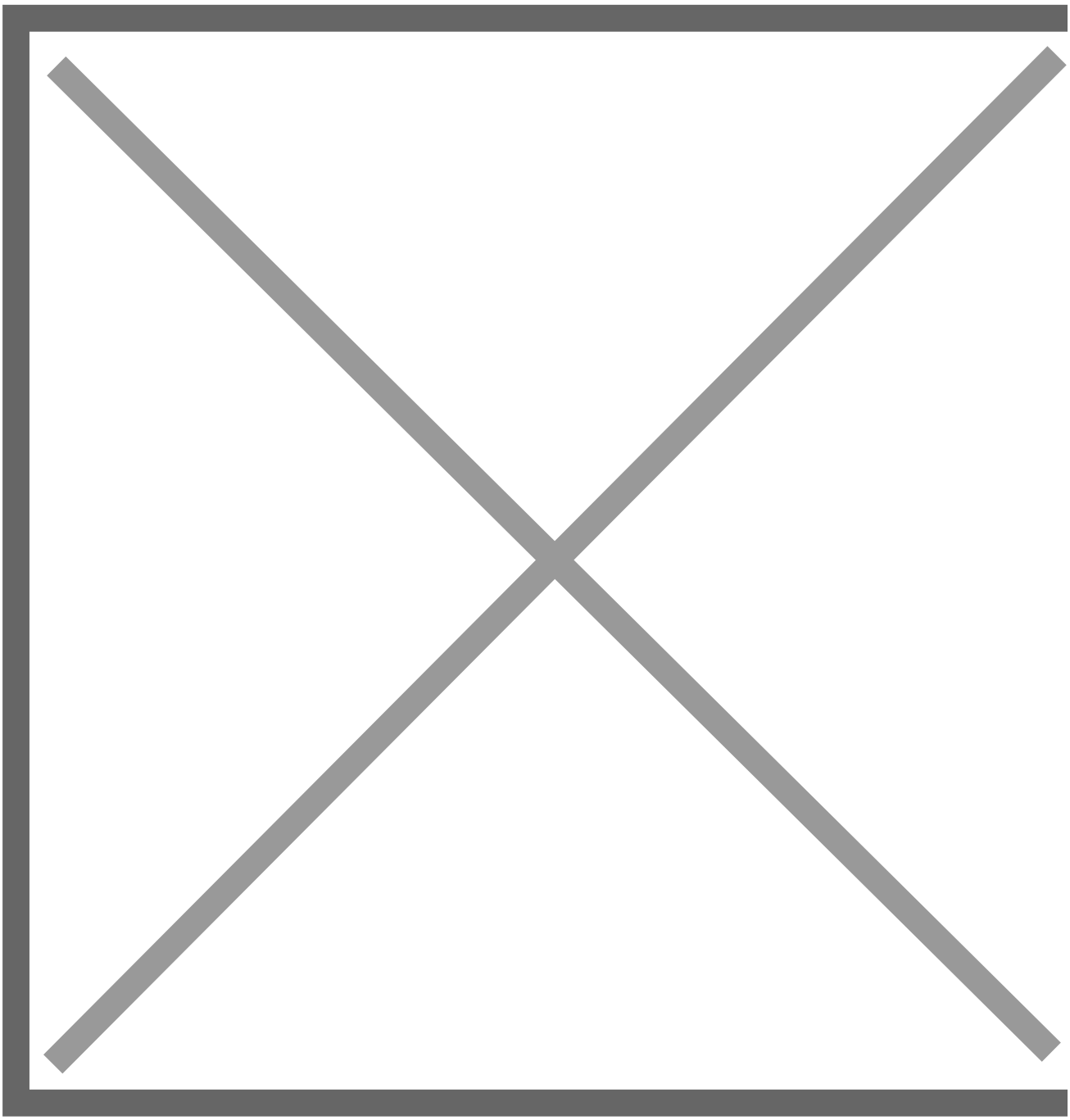
Von Jan Mahn

c't 7/2023 S. 164

## kompakt

- Kubernetes kann über eine Schnittstelle namens „Container Storage Interface“ (CSI) verschiedene Anbieter von Speicherplatz anbinden. Ein solcher ist Longhorn.
- Longhorn speichert Volumes nicht nur redundant, sondern fertigt auch Snapshots und inkrementelle Backups.
- Für die Verwaltung gibt es eine Weboberfläche, die auf einen Blick verrät, wie es um den Cluster bestellt ist.

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 ein einem fünfteiligen Tutorial beschrieben [1, 2, 3, 4, 5]. 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 [4]). 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/yqtp), 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 (sogenannten Replicas) 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 [3]. 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/yqtp).

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 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 --versior
```

Wenn Sie diesen Artikel deutlich nach März 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/yqtp 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.

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

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 Abschnitt „Ein Volume, bitte“. Wenn Sie für die Administration des Clusters das GUI von Lens nutzen (zum Download über ct.de/yqtp), 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 in [4] 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.

## 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: |
    YWRtaW46JGFwcjEkSVdYUnZWQUkkb0d...
```

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
```

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:

```
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 www.example.org/longhorn/ erreichbar. Einziger kleiner Schönheitsfehler: Der / am Ende ist Pflicht, sonst lädt die Seite nicht.

# 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 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:
   - 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.

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

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/yqtp).

# 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 [6] 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, die Longhorn mitbringt – den schnellen Überblick über Füllstände 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.

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 sogenannter 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. Dort findet sich ein Button, um einen Snapshot zu

starten.

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.

# 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 – so wie man es von einem redundanten Cluster erwartet.

# Alternativen

Longhorn ist längst nicht der einzige Storage-Anbieter, der an das CSI von Kubernets 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 PersistentVolumeClaims.

# Horizont

Mit Longhorn kennen Sie nun einen Storage-Anbieter, der eine gute Figur macht, wenn Sie sich als Kubernetes-Administrator auch um Ihren eigenen redundanten Speicherplatz kümmern müssen. Mit dem CSI kennen Sie ein zentrales Konzept, um Kubernetes für eigene Bedürfnisse zu erweitern. Ähnlich wie CSI funktionieren zwei andere Arten von Schnittstellen, das Container-Runtime-Interface (CRI) und das Container-Network-Interface (CNI). Per CRI kann man andere Container-Runtimes anbinden (eher keine Aufgabe, die man im Alltag braucht), das CNI braucht man immer dann, wenn eine Software tiefer in den Netzwerkverkehr im Cluster eingreifen muss. (*jam@ct.de*)

## 1. Literatur


2. Jan Mahn, Containerkompetenzoffensive, Auf dem Lernpfad zum Kubernetes-Kenner, Teil 1, c’t 22/2022, S. 164
3. Jan Mahn, Dickschiffkapitän, Auf dem Lernpfad zum Kubernetes-Kenner, Teil 2, c’t 23/2022, S. 158
4. Jan Mahn, Containervernetzer, Auf dem Lernpfad zum Kubernetes-Kenner, Teil 3, c’t 25/2022, S. 162
5. Jan Mahn, Containerladeoffizier, Auf dem Lernpfad zum Kubernetes-Kenner, Teil 4, c’t 26/2022, S. 130

6. Jan Mahn, Container-Sicherheitsbegehung, Auf dem Lernpfad zum Kubernetes-Kenner, Teil 5, c’t 3/2023, S. 154
7. Jan Mahn, Gemeinsame Wahrheit, Wie verteilte Systeme dank Raft-Algorithmus zusammenarbeiten, c’t 21/2022, S. 146

**Dokumentation:** ct.de/yqtp