Serverexperimente I

Dokumentiert meine Experimente mit verschiedenen Servern

    - 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

- Raspberry PI 3 mit Docker
    - Docker installieren

- Automatisierung mit Ansible
    - Telerec't
    - Ordnung im Königreich
    - Schaltzentrale für den Server

- Git basics
- Node.JS und Angular
- 1blu KVM-Installation
    - Installation von Ubuntu auf einem vServer via KVM

- Datenbanken
    - PostgreSQL mit homebrew

Hetzner vServer bestellen & sichern

Beschreibt die Bestellung und Absicherung eines Hetzner vServers mit Ubuntu 20.04/22.04 LTS.

Bestellung

Über die Hetzner Cloud Console kann ein neuer Server geordert oder aus einem Snapshot erstellt werden. In der Regel werden aus Kostengründen die neuen ARM-Server bevorzugt, aber noch ist die Softwarewelt nicht so weit, ARM CPUs in allen Bereichen zu unterstützen. Für den Fall der Fälle kann auf AMD- oder Intel-Hardware ausgewichen werden.

Ein üblicher Standard-Server sieht etwa wie folgt aus:

Standort	Nürnberg, Falkenstein oder Helsinki
Image	Ubuntu 22.04
Typ	Shared CPU Arm64 (Ampere) CAX11 (2 Cores, 4 GB RAM, 40 GB SSD, 20 TB Traffic)
Networking	IP4 optional / IP6 optional / Private Network optional (je nach Bedarf)
Platzierungsgruppen	Empfehlenswert, wenn mehrere Server im Projekt laufen (z. B. Kubernetes)
SSH-Keys	Konfigurierte Public Keys von allen administrierenden Systemen (empfohlen)
Firewalls	Je nach Einsatz - für öffentliche Server ein Muss!
Backups	täglich, 7 Tage Zyklus. Empfehlenswert - Kosten: 20% vom Serverpreis
Name	nach eigenem Ermessen
Kosten	5,29€ / Monat mit IP4-Adresse und Backups aktiviert (Stand April 2024)

Nach Abschluss einer Bestellung wird der Server innerhalb weniger Sekunden erstellt und seine Basisdaten sowie seine IP-Adresse(n) angezeigt. Mit der IP-Adresse kann sich mittels ssh auf ihm angemeldet werden:

```
ssh root@<IP-Adresse>
```

Eine IPv6 Adresse wird nicht ganz korrekt angezeigt. Am Ende erscheint ein ::/64, was bedeuten würde das das letzte Paket 0000 lautet. Dem ist nicht so, denn das letzte Paket des Adressblocks lautet korrekt 0001.
Um sich die sperrige IP-Adresse nicht merken zu müssen, empfiehlt es sich, einen passenden Namen mit der IP-Adresse in Verbindung zu bringen und in der Shell zu exportieren. Für einen dauerhaften Export benötigt es nur einen Eintrag in der Datei .zshrc (oder .bashrc):

```
export myserver=<IP-Adresse>
```

Damit ist dann der ssh Verbindungsaufbau mit dem Befehl ssh root@$myserver möglich.

Der Login erfordert dank des übergebenen öffentlichen SSH-Key kein Passwort, es sein denn, der ssh-Key selbst ist geschützt. Von Hetzner wird bei Angabe eines ssh-Keys kein Password generiert und per E-Mail verschickt.

Absicherung

Externe Server bzw. generell alle Server, die am Internet hängen, sollten Anmeldungen auf der Konsole ausschließlich über einen SSH-Schlüssel zulassen. Zudem sollte sich der User root gar nicht von außen anmelden können. Außerdem sollte eine Firewall Zugriffe nur über explizit erlaubte Ports erlauben und eine Software installiert sein, die Angriffsversuche erkennt und blockiert. Am Beispiel eines Hetzner Cloud vServers unter Ubuntu 22.04 LTS wird hier beschrieben, wie ein solcher Server abgesichert werden kann.

Anlegen eines neuen Users mit root Rechten

Erfreulicherweise können bei Hetzner, wie auf der Seite Hetzner vServer bestellen & absichern beschrieben, in der Bestellung eines Servers bereits alle SSH-Schlüssel, die auf dem Server erlaubt sein sollen, mit angegeben werden. Voraussetzung dafür ist, dass die öffentlichen Schlüssel im Hetzner-Account abgelegt wurden. Diese Option sollte unbedingt genutzt werden, denn so ist der Server gleich von Beginn an mit einem Grundschutz gegen Attacken von außen versehen, denn eine Anmeldung als root ohne einen passenden privaten Schlüssel ist nicht mehr möglich.

Nachdem der Server das erstmals gestartet wurde, erfolgt die Anmeldung mit `ssh root@<ip_adresse>`. Dazu ist es unbedingt erforderlich, dass ein gültiger öffentlicher Schlüssel im .ssh Verzeichnis des aufrufenden Users liegt. Dafür hat Hetzner bei der Anlage des Servers gesorgt. Alle in der Bestellung angegebenen öffentlichen Schlüssel sind in der Datei ~/.ssh/authorized_keys eingetragen.

Nach einer erfolgreichen Anmeldung wird als User root auf dem Server gearbeitet. Aus diesem Grund enthalten die nachfolgend aufgeführten Kommandos auch kein vorangestelltes sudo.

Als erste Aktion nach einer Neuinstallation und der Erstanmeldung als root ist unbedingt das Betriebssystem zu aktualisieren. Dazu ist der Befehl `apt update && apt upgrade -y` auszuführen. Unter Umständen weist das Upgrade am Ende darauf hin, dass ein Reboot des Systems erforderlich ist. Das kann schnell mit dem Befehl `reboot` herbeigeführt werden.

Im nächsten Schritt ist die Firewall zu konfigurieren und aktivieren. Die Firewall ufw ist bereits installiert, aber noch nicht aktiviert. Bevor sie aktiviert werden kann, sind allerdings noch ein paar Konfigurationen erforderlich. Eingehender Verkehr ist nur auf den explizit zugelassenen Ports zulässig, ausgehender Verkehr wird generell erlaubt.

```
# Eingehenden Verkehr generell verbieten
sudo ufw default deny incoming
```

```
# Ausgehenden Verkehr generell eerlauben
sudo ufw default allow outgoing

# SSH Verbindungen zulassen
sudo ufw allow ssh

# Wenn auch HTTP/HTTPS erlaubt sein soll
sudo ufw allow http
sudo ufw allow https

# Firewall aktivieren
sudo ufw enable
```

Im nächsten Schritt ist ein User mit sudo Rechten einzurichten:

```
#
# Benutzer 'sysadmin' einrichten und zum sudoer machen.
# Dabei wird vorausgesetzt, dass es sich beim eingelogten
# user root und dem neuen user sysadmin um ein und dieselbe
# Person handelt.
#
adduser sysadmin
usermod -a -G sudo sysadmin
mkdir /home/sysadmin/.ssh
cp ~/.ssh/authorized_keys /home/sysadmin/.ssh/
chmod 700 /home/sysadmin/.ssh
chmod 400 /home/sysadmin/.ssh/authorized_keys
chown sysadmin:sysadmin /home/sysadmin/.ssh
chown sysadmin:sysadmin /home/sysadmin/.ssh/authorized_keys
```

In einem neuen Terminal sollte nun ein ssh Anmeldeversuch mit dem neuen User erfolgen. Die Anmeldung muss ohne Eingabe eines Passworts gelingen, es sei denn, das System fragt nach dem Passwort für den ssh Key, wenn dieser mit einem Passwort abgesichert ist.

Nach erfolgreicher Anmeldung muss abschließend geprüft werden, ob der neue User sysadmin mittels sudo mit root-Rechten arbeiten kann. Um dem User zukünftig die Passworteingabe nach einem sudo Befehl zu ersparen, führen folgende Schritte zum Ziel:

```
#
# Eine Datei im Verzeichnis /etc/sudoers.d anlegen
#
```

```
sudo vi /etc/sudoers.d/sysadmin-user

#
# In dieser Datei folgende Zeile einfügen:
#
sysadmin ALL=(ALL) NOPASSWD:ALL

#
# In einem neuen Terminalfenster testen:
#
sudo -i
```

> Der Verzicht auf das sudo Password ist eine Schwächung der Systemsicherheit!

> Die Nachfolgenden Schritte dürfen nur durchgeführt werden, wenn sich der neue User per ssh ohne Password mit seinem ssh Key anmelden und root Rechte einnehmen kann. Ansonsten droht der Ausschluss aus dem Server!.

Anmeldung des Users root verbieten

Ein nächster wichtiger Schritt hin zu einem sicheren System ist das Unterbinden des Login als User root und des Login mit Passwort. Ein Login soll nur noch mittels ssh Schlüssel möglich sein. Dazu sind in der Datei */etc/ssh/sshd_config*folgende Einträge wie dargestellt zu ändern:

```
#
# Anmelden mit root verbieten und nur publickey erlauben
#
PermitRootLogin no
AuthenticationMethods publickey
PasswordAuthentication no
```

Nach der Änderung muss der ssh Service mit dem Befehl `sudo systemctl restart sshd` neu gestartet werden.

Sicherheit

Sicherheit

Sicherheit: TLS-Protokoll absichern

In der Basiskonfiguration erlaubt der Server viele mittlerweile als unsicher eingestufte Protokolle und Cipher. Zudem wird nicht zwingend eine Transport-Sicherheit verlangt. Ein Artikel in der c't 2/2022 zu diesem Thema (Security: TLS optimieren) veranschaulicht die Risiken und gibt Lösungsvorschläge an die Hand. besonders erwähnenswert sind dabei die Tools für den Sicherheitschek der Domain (https://ssllabs.com/ssltest) und die Konfigurationsunterstützung (https://ssl-config.mozilla.org).

Im Wesentlichen sind im Falle des V-Servers mit Ubuntu 20.04 und einem Zertifikat von LetsEncrypt für die Domain mykoelle.de zwei Dateien der Apache Konfiguration anzupassen das Apache Modul Headers aktivieren, damit der Server die begehrte A+ Bewertung von im Sicherheitscheck erhält.

Bei der ersten Datei handelt es sich um die vom LetsEncrypt certbot erstellte Datei */etc/letsencrypt/options-ssl-apache.conf*. Durch die Änderung dieser Datei wird sie nicht mehr durch den certbot aktualisiert. Daher muss regelmäßig das Log des certbot (/var/log/letsencrypt/letsencrypt.log) auf updates für diese Datei überprüft werden. Hie die komplette Datei mit Stand 06.01.2022:

```
# This file contains important security parameters. If you modify this file
# manually, Certbot will be unable to automatically provide future security
# updates. Instead, Certbot will print and log an error message with a path to
# the up-to-date file that you will need to refer to when manually updating
# this file.

SSLEngine on

# Intermediate configuration, tweak to your needs

#-------------------------
# Original Cipher configuguration
#
#SSLProtocol             all -SSLv2 -SSLv3
```

```
#SSLCipherSuite          ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-
ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-
AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-
SHA256:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-
AES128-SHA:ECDHE-ECDSA-AES256-SHA384:ECDHE-ECDSA-AES256-SHA:ECDHE-RSA-AES256-SHA:DHE-RSA-
AES128-SHA256:DHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA256:DHE-RSA-AES256-SHA:ECDHE-ECDSA-DES-
CBC3-SHA:ECDHE-RSA-DES-CBC3-SHA:EDH-RSA-DES-CBC3-SHA:AES128-GCM-SHA256:AES256-GCM-
SHA384:AES128-SHA256:AES256-SHA256:AES128-SHA:AES256-SHA:DES-CBC3-SHA:!DSS
#SSLHonorCipherOrder     on
#SSLCompression          off
#
#--------------------------

#--------------------------
# Changed (see ssl-config.mozilla.org and ssllabs.com/ssltest)
#
SSLProtocol             all -SSLv2 -SSLv3 -TLSv1 -TLSv1.1
SSLCipherSuite          ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-
AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-
CHACHA20-POLY1305:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384
SSLHonorCipherOrder     off
SSLSessionTickets       off
Protocols               h2 http/1.1
Header                  always set Strict-Transport-Security "max-age=63072000"
#
#--------------------------

SSLOptions +StrictRequire

# Add vhost name to log entries:
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-agent}i\"" vhost_combined
LogFormat "%v %h %l %u %t \"%r\" %>s %b" vhost_common

#CustomLog /var/log/apache2/access.log vhost_combined
#LogLevel warn
#ErrorLog /var/log/apache2/error.log
```

Damit im Header eine zwingende Transport-Sicherheit eingetragen werden kann, muss das Modul Headers des Apache Webservers aktiviert werden. Dies geschieht mit dem Befehl `sudo a2enmod headers`.

Für eine Optimierung sorgen die beiden folgenden Zeilen in der Datei */etc/apache2/sites-enabled/bookstack-le-ssl.conf*. Sie sind unterhalb des </VirtualHost> Tags, also außerhalb der VirtuaHost Beschreibung einzufügen:

```
<VirtualHost *:443>
....
</VirtualHost>
#
# Added (see ct 2/2022 Security: TLS optimieren und ssl-config.mozilla.org and
ssllabs.com/ssltest)
#
SSLUseStapling          On
SSLStaplingCache        "shmcb:logs/ssl_stapling(32768)"
```

Nun kann mit den Anweisungen `sudo apachectl configtest` und `sudo apachectl restart` der Apache Webserver neu gestartet werden. Ein Test der Domain via https://ssllabs.com/ssltest sollten nun ein grünes A+ ergeben.T

Sicherheit

Eigene CA und lokales Zertifikat erstellen

Für Tests ist es sinnvoll, Serveranwendungen auch unter https zu testen. Um das auf dem Entwicklungsrechner zu tun, benötigt man ein lokales Zertifikat und möglichst auch eine eigene CA, damit das Zertifikat auch ohne die lästigen Warnungen akzeptiert wird.

Auf einem Mac oder jedem anderen System mit Homebrew kann das mit brew erledigt werden. Zuerst aktualisiert man brew:

```
brew update && brew upgrade
```

Danach installiert man das Tool mkcert:

```
brew install mkcert
```

Damit mkcert das neu erstellte Zertifikat bzw. die CA in Firefox installieren kann, benötigt mkcert noch das Tool certutil aus dem Cask nss. Wird also auch mit Firefox getestet, ist nss noch zu installieren:

```
brew install nss
```

Nun kann eine CA und ein Zertifikat für localhost erstellt werden:

```
mkcert -install
mkcert localhost 127.0.0.1 ::1
```

Der Befehl `mkcert -install` kann übrigens beliebig oft wiederholt werden.

Um die erzeugten <name>.pem- und <name-key>.pem Dateien in <name>.crt und <name>.key Dateien umzuwandeln, wird openssl genutzt.

```
openssl x509 -in localhost.pem -out localhost.crt
openssl pkey -in localhost-key.pem -out localhost.key
```

Einsatz z. B. in einer Go-Anwendung mit dem Echo Framework:

```
func main() {
	e := echo.New()
	// add middleware and routes
	// ...

	sc := echo.StartConfig{Address: ":1323"}
	if err := sc.StartTLS(context.Background(), e, "localhost.crt", "localhost.key"); err != nil {
		e.Logger.Error("failed to start server", "error", err)
	}
}
```

Docker mit Traefik

Hier geht es um Experimente mit Docker Traefik. Geplant war ein kleiner Server, der einen reinen HTML-Blog (Hugo), eine Wissensdatenbank (BookStack) und einen WebDav-Server zur Verfügung stellt. Alle Services sollten als Docker Services installiert werden und nur über https und Subdomains ansprechbar sein. Die Zertifikatsverwaltung sollte automatisiert erfolgen. Also das ideale Einsatzgebiet für Traefik.

Die Basis

Warum tue ich das?

Mit Traefik hatte ich zu Beginn meiner Docker-Experimente schon einmal gearbeitet. Ursache war ein Artikel in der Zeitschrift c't, die die Möglichkeiten von Traefik einmal angerissen haben.
Auf die Idee, ein neues Traefik-Experiment anzugehen, brachte mich eine Information meines Lieblings-Cloud-Hosters Hetzner. Dort gibt es seit kurzem Server auf Basis der ARM64 Architektur zu mieten. Als Besitzer eines der MacBooks Pro mit Apple M1 Max Chip, der auf eben dieser ARM-Architektur basiert, fand ich das Thema spannend. Zumal ich gelernt habe, dass gerade im Docker Umfeld lange nicht alles als ARM-Ready zu bezeichnen ist. Und das, obwohl ARM-CPUs nicht nur im Serversektor auf enorme Zuwachsraten schauen können.

Welche Ziele möchte ich erreichen?

Um dem Experiment einen tieferen Sinn zu geben, habe ich mir Ziele gesteckt, die bei erfolgreicher und zufriedenstellender Umsetzung zu einer Umgestaltung meiner derzeitigen Web-Infrastuktur führen könnten. Konkret lauten diese:

- Der Server soll über die Internet-Protokolle IPv4 und IPv6 angesprochen werden können.
- Der Server wird standardmäßig abgesichert.
- Es werden ausschließlich sichere und verschlüsselte Protokolle (HTTPS, DAVS, ...) verwendet.
- Notwendige Zertifikate sollen über Let's Encrypt automatisch bezogen und verwaltet werden.
- Auf dem Server läuft neben den für das Betriebssystem erforderlichen Anwendungen ausschließlich Docker.
- Traefik übernimmt die Zertifikatsverwaltung und Funktion des Proxy.
- Das Traefik-Dashbord ist über die Subdomain traefik.domain.tld erreichbar und mit BasicAuth gesichert.
- Die offizielle Webseite, ein mit hugo erstellter reiner HTML-Blog, wird über Nginx bereitgestellt.
- Die offizielle Webseite ist über https://domain.tld oder https://www.domain.tld gleichermaßen erreichbar.
- Eine WebDav-Anbindung ist über die Subdomain davs.domain.tld erreichbar.
- Eine BookStack-Instanz ist über die Subdomain book.domain.tld erreichbar.
- Die für BookStack notwendige Datenbank liegt in einem eigenen Netzwerk und ist von außen nicht zu erreichen.

Der erste Schritt - Server anlegen

Also gespannt ans

Traefik Reverse Proxy

Verzeichnisse und Dateien anlegen

```
mkdir -p /opt/containers/traefik/data
touch /opt/containers/traefik/data/acme.json
chmod 600 /opt/containers/traefik/data/acme.json
touch /opt/containers/traefik/data/traefik.yml
```

> Wichtig: Nicht vergessen, die Datei acme.json (oder acme_letsencrypt.json) anzulegen

./docker-compose.yml erstellen

```
#
# /opt/containers/trefik/docker-compose.yml
# Reverse Proxy Traefik installieren
#
version: '3.9'
services:
  traefik:
    container_name: traefik
    image: traefik:latest
    volumes:
      # Traefik mit der Zeit vom Server syncronisieren
      - /etc/localtime:/etc/localtime:ro
      # Lesezugriff auf den UNIX Docker socket
      - /var/run/docker.sock:/var/run/docker.sock:ro
      # Traefik-Konfiguration bereitstellen
      - ./data/traefik.yml:/traefik.yml:ro
      # Datei mit Zertifikaten bereitstellen
      - ./data/acme_letsencrypt.json:/acme_letsencrypt.json
      # Datei mit dynamischer Konfiguration bereitstellen
      - ./data/dynamic_conf.yml:/dynamic_conf.yml
    labels:
      # Traefik durch Watchtower aktualisieren lassen
```

```
      # https://goneuland.de/docker-images-automatisiert-aktualisieren-mit-watchtower/
      - "com.centurylinklabs.watchtower.enable=true"
      # Traefik für Traefik aktivieren
      - "traefik.enable=true"
      # Als Einstiegspunkt wählen wir direkt HTTPS
      - "traefik.http.routers.traefik.entrypoints=https"
      # Domain anpassen unter der Traefik erreichbar sein soll.
      - "traefik.http.routers.traefik.rule=Host(`traefik.die-steffens.eu`)"
      # Middlewares definieren, welche verwendet werden sollen.
      # Hier die Authentifizierung via htpasswd + alle die unter default definiert sind.
      - "traefik.http.routers.traefik.middlewares=traefikAuth@file,default@file"
      # SSL Zertifikat abrufen
      - "traefik.http.routers.traefik.tls=true"
      # Definierten Zertifikat Resolver aus traefik.yml wählen
      - "traefik.http.routers.traefik.tls.certresolver=http"
      - "traefik.http.routers.traefik.service=api@internal"
      - "traefik.http.services.traefik.loadbalancer.server.port=80"
      - "traefik.http.services.traefik.loadbalancer.sticky.cookie.httpOnly=true"
      - "traefik.http.services.traefik.loadbalancer.sticky.cookie.secure=true"
     # Traefik dem Proxy-Netzwerk hinzufügen.
      - "traefik.docker.network=proxy"
    restart: unless-stopped
    security_opt:
      - no-new-privileges:true
    networks:
      proxy:
    hostname: traefik
    ports:
      # Ports definieren, welche durch Traefik gemanaget werden.
      - "80:80"
      - "443:443"

networks:
  proxy:
    name: proxy
    driver: bridge
    attachable: true
```

./data/dynamic_conf.yml füllen

```
#
# /opt/containers/trafik/data/dynamic_conf.yml
#
# TLS
# Hier werden alle notwendigen Einstellungen für das Zertifikat getroffen.
# In Kombination mit den Einstellungen unter http.middlewares.default-security-headers
bekommen  wir ein A+ Zertifikat.
tls:
  options:
    default:
      minVersion: VersionTLS12
      cipherSuites:
        - TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
        - TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
        - TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305
        - TLS_AES_128_GCM_SHA256
        - TLS_AES_256_GCM_SHA384
        - TLS_CHACHA20_POLY1305_SHA256
      curvePreferences:
        - CurveP521
        - CurveP384
      sniStrict: true

# Middlewares
# Optionale Optimierungen, die bei jeder Anfrage vorgenommen werden sollen bevor diese an den
Zielcontainer geleitet wird.
http:
  middlewares:
    # Eine grundlegende Authentifizierungs-Middleware, um das Traefik-Dashboard via htpasswd
zu schützen
    # Um die Authentifizierung für einen Container zu nutzen können wir
"traefik.http.routers.definierteRoute.middlewares=traefikAuth@file" nutzen
    traefikAuth:
      basicAuth:
        users:
          - "Kamikaze-Jim:$apr1$P/tG13aC$BdXjI6AtKysGWvRAr9aoI0"

    # Empfohlene Standard-Middleware für die meisten Dienste
    # Hinzufügbar via "traefik.http.routers.definierteRoute.middlewares=default@file"
    # Equivalent mit "traefik.http.routers.definierteRoute.middlewares=default-security-
```

```
headers@file,gzip@file"
    # Die Liste kann hier auch beliebig erweitert werden
    default:
      chain:
        middlewares:
          - default-security-headers
          - gzip

    # Kompatibilität zu alten Anleitungen. Damit kann auch wieder
"traefik.http.routers.definierteRoute.middlewares=secHeader@file"
    secHeaders:
      chain:
        middlewares:
          - default-security-headers
          - gzip

    # Standard Header
    default-security-headers:
      headers:
        browserXssFilter: true
        contentTypeNosniff: true
        forceSTSHeader: true
        frameDeny: true
#       Deprecated
#       sslRedirect: true
        #HSTS Configuration
        stsIncludeSubdomains: true
        stsPreload: true
        stsSeconds: 31536000
        customFrameOptionsValue: "SAMEORIGIN"
    # Gzip Kompression
    gzip:
      compress: {}
```

User und Passwort festlegen

```
#
# Erzeugt das für obige Datei erforderliche USR/PWD-Gespann
#
echo $(htpasswd -nb <user> '<password>')
```

```
#
# Zum Beispiel:
# echo $(htpasswd -nb jonathan 'supersicher')
# Ausgabe: jonathan:$$apr1$$J8vBlNIm$$lSwLv9iGa8KcCct3EyLD41
# Sollte es an dieser Stelle zu Problemen kommen kann man sich auch das Passwort auf Webseiten
# wie zum Beispiel: https://www.redim.de/blog/passwortschutz-mit-htaccess-einrichten
generieren lassen.
```

./data/traefik.yml füllen

```
#
# /opt/containers/traefik/data/traefik.yml
#
# Statische Traefik-Konfigurationsdatei
# https://doc.traefik.io/traefik/getting-started/configuration-overview/#the-static-
configuration
# https://doc.traefik.io/traefik/reference/static-configuration/cli/

api:
  dashboard: true                           # Aktivieren des Dashboard

# Certificate Resolver
# Diese sind für den Abruf von Zertifikaten von einem ACME-Server zuständig
# https://doc.traefik.io/traefik/https/acme/#certificate-resolvers
certificatesResolvers:
  http:
    acme:
      email: "ralf@rastef.de"               # E-Mail-Adresse für die Registrierung
      storage: "acme_letsencrypt.json"      # Datei für die Speicherung von Zertifikate (Ich
weiche hier bewusst von dem "Standard": acme.json ab)
      httpChallenge:
        entryPoint: http

# EntryPoints
# EntryPoints sind die Netzwerk-Eingangspunkte in Traefik. Sie definieren den Port, der die
Pakete empfängt.
# https://doc.traefik.io/traefik/routing/entrypoints/
entryPoints:
  http:
    address: ":80"                          # Erstellen des Einstiegspunkt für HTTP (Port
```

```
80)
    http:
      redirections:                          # Weiterleitung von HTTP auf HTTPS (Port 80 zu
Port 443).
        entryPoint:
          to: "https"                        # Das Ziel
          scheme: "https"                    # Umleitungszielschema
  https:
    address: ":443"                          # Erstellen des Einstiegspunkt für HTTPS (Port
443)



global:
  checknewversion: true                      # In regelmäßigen Abständen prüfen, ob eine neue
Version veröffentlicht wurde.
  sendanonymoususage: false                  # Regelmäßige Übermittlung anonymer
Nutzungsstatistiken.

providers:
  docker:
    endpoint: "unix:///var/run/docker.sock"   # Den UNIX Docker socket beobachten
    exposedByDefault: false                   # Nur Container ausstellen, die explizit
aktiviert sind (mit dem Label traefik.enabled)
    network: "proxy"                          # Standardnetzwerk, das für Verbindungen zu
allen Containern verwendet wird.
  file:
    filename: "./dynamic_conf.yml"            # Link zur dynamischen Konfiguration
    watch: true                               # Achten auf Änderungen
  providersThrottleDuration: 10               # Frequenz in welchen Abständen die
Konfiguration nachgeladen wird
```

Die eigentliche Anleitung stammt von hier: https://goneuland.de/traefik-v2-reverse-proxy-mit-crowdsec-einrichten/

Webserver mit nginx

Um einen nginx Webserver Container einzubinden, sind folgende Schritte durchzuführen:

- Anlegen des Verzeichnis ***/opt/containers/nginx***
- Anlegen des Verzeichnis ***/opt/containers/nginx/html***

```
sudo mkdir -p /opt/containers/nginx/html
```

Für den Containerstart ist die Datei ***/opt/containers/nginx/docker-compose.yml*** mit folgendem Inhalt anzulegen:

```
#
# /opt/containers/nginx/docker-compose.yml
# Startet einen nginx Webserver
#
version: '3'
services:
  nginx:
    image: nginx:latest
    container_name: nginx
    restart: unless-stopped
    networks:
      - proxy
    volumes:
      - ./html:/usr/share/nginx/html
    labels:
      - "traefik.enable=true"
      - "traefik.http.middlewares.redirect2www.redirectregex.regex=^https://die-
steffens.eu/(.*)"
      - "traefik.http.middlewares.redirect2www.redirectregex.replacement=https://www.die-
steffens.eu/$${1}"
      - "traefik.http.routers.nginx-www.entrypoints=https"
      - "traefik.http.routers.nginx-www.rule=Host(`www.die-steffens.eu`,`die-steffens.eu`)"
      - "traefik.http.routers.nginx-www.tls=true"
      - "traefik.http.routers.nginx-www.tls.certresolver=http"
      - "traefik.http.routers.nginx-www.service=nginx"
```

```
      - "traefik.http.routers.nginx-www.middlewares=redirect2www,secHeaders@file"
      - "traefik.http.services.nginx.loadbalancer.server.port=80"
      - "traefik.docker.network=proxy"
networks:
  proxy:
    external: true
```

Der Webserver kann über das unsichere Port 80 Protokoll http oder das sichere Port 443 Protokoll https angesprochen werden. In ersterem Fall wird auf das sichere https Protokoll umgelenkt.

Gestartet und gestoppt wird wie üblich mit:
`/opt/containers/bookstack/docker compose -f /opt/containers/nginx/docker-compose.yml up -d`
`/opt/containers/bookstack/docker compose -f /opt/containers/nginx/docker-compose.yml down`

Bookstack

Um einen Bookstack Container einzubinden, sind folgende Schritte durchzuführen:

- Anlegen des Verzeichnis ***/opt/containers/bookstack***
- Anlegen des Verzeichnis ***/opt/containers/bookstack/data/db***
- Anlegen des Verzeichnis ***/opt/containers/bookstack/data/bs***

```
sudo mkdir -p /opt/containers/bookstack/data/db
sudo mkdir /opt/containers/bookstack/data/bs
```

Für den Containerstart ist die Datei ***/opt/containers/bookstack/docker-compose.yml*** mit folgendem Inhalt anzulegen:

```
#
# /opt/containers/bookstack/docker-compose.yml
# Ein Bookstack Container mit MariaDB
#
version: '3'
services:
  db:
    image: lscr.io/linuxserver/mariadb:latest
    container_name: bookstack-db
    networks:
      - db-network
    environment:
      - TZ=Europe/Berlin
      - MYSQL_ROOT_PASSWORD=RaRuRickZaubertrick
      - MYSQL_DATABASE=bookstack
      - MYSQL_USER=bookstack
      - MYSQL_PASSWORD=DaKommstDuNieDrauf
    volumes:
      - ./data/db:/config
    restart: unless-stopped
    labels:
      - traefik.enable=false
```

```
  app:
    image: lscr.io/linuxserver/bookstack:latest
    container_name: bookstack-app
    networks:
      - db-network
      - proxy
    depends_on:
      - db
    environment:
      - APP_URL=https://book.die-steffens.eu
      - DB_HOST=db
      - DB_PORT=3306
      - DB_DATABASE=bookstack
      - DB_USERNAME=bookstack
      - DB_PASSWORD=DaKommstDuNieDrauf
    restart: unless-stopped
    volumes:
      - ./data/bs:/config
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.bookstack.entrypoints=https"
      - "traefik.http.routers.bookstack.rule=Host(`book.die-steffens.eu`)"
      - "traefik.http.routers.bookstack.tls=true"
      - "traefik.http.routers.bookstack.tls.certresolver=http"
      - "traefik.http.routers.bookstack.service=bookstack"
      - "traefik.http.routers.bookstack.middlewares=secHeaders@file"
      - "traefik.http.services.bookstack.loadbalancer.server.port=80"
      - "traefik.docker.network=proxy"


networks:
  db-network:
    external: false
  proxy:
    external: true
```

Es werden zwei Container gestartet: Die Datenbank MariaDB und der Bookstack Container. Für die Kommunikation mit der Datenbank wird ein internes Netzwerk eingerichtet, das von Außen nicht zu erreichen ist.

Gestartet und gestoppt wird wie üblich mit:
`/opt/containers/bookstack/docker compose -f /opt/containers/bookstack/docker-compose.yml up -d`

```
/opt/containers/bookstack/docker compose -f /opt/containers/bookstack/docker-compose.yml down
```

WebDAV Container

Um einen WebDAV Zugang über ein gesichertes Protokoll bereitzustellen, sind nur wenige Schritte erforderlich:

- Das Verzeichnis ***/opt/containers/webdav*** muss angelegt werden.
- Das Verzeichnis ***/opt/containers/webdav/dat***a muss angelegt werden.
- Das Verzeichnis ***/opt/containers/webdav/config*** muss angelegt werden.
- Das Verzeichnis ***/opt/containers/webdav/data*** soll für alle zu lesen und zu beschreiben sein (***chmod 777***).

```
sudo mkdir -p /opt/containers/webdav/config
sudo mkdir /opt/containers/webdav/data
sudo chown 777 /opt/containers/webdav/data
```

Im Verzeichnis ***opt/containers/webdav*** ist die nachfolgend aufgezeigte Datei ***docker-compose.yml*** zu erstellen:

```
#
# /opt/containers/webdav/docker-compose.yml
# Starts and initializes a WebDAV Container
#
version: '3'
services:
  webdav:
    image: chonjay21/webdav:latest
    container_name: webdav
    restart: unless-stopped
    environment:
      APP_USER_NAME: meganeKAR8879
      APP_USER_PASSWD: qAJsPWCc7YcMro2vfRWHv2tfad2hh2kfWcieMJ2VCmXKT3ggNMRKVLQTSrsEcUNT
      SERVER_NAME: davs.die-steffens.eu
      APP_UID: 1000
      APP_GID: 1000
      APP_UMASK: 007
      TZ: Europe/Berlin
      FORCE_REINIT_CONFIG: true
```

```
    networks:
      - proxy
    security_opt:
      - no-new-privileges:true
    volumes:
      - ./data:/var/webdav/data
      - ./config:/webdav/config
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.webdav-secure.entrypoints=https"
      - "traefik.http.routers.webdav-secure.rule=Host(`davs.die-steffens.eu`)"
      - "traefik.http.routers.webdav-secure.tls=true"
      - "traefik.http.routers.webdav-secure.tls.certresolver=http"
      - "traefik.http.routers.webdav-secure.service=webdav"
      - "traefik.http.services.webdav.loadbalancer.server.port=80"
      - "traefik.docker.network=proxy"
      - "traefik.http.routers.webdav-secure.middlewares=secHeaders@file"
networks:
  proxy:
    external: true
```

Mit `docker compose -f /opt/containers/webdav/docker-compose.yml up -d` kann der Container gestartet werden.
Mit `docker compose -f /opt/containers/webdav/docker-compose.yml down` kann der Container gestoppt werden.

Das für alle User offene Datenverzeichnis ist natürlich für komplexere Szenarien mit Benutzenden in unterschiedlichen Rollen nicht zu empfehlen. Als Familiensammelverzeichnis ist es allerdings ausreichend. Weitere Informationen finden sich in der Beschreibung des Containers.

Docker ohne Traefik (ubusrv)

Einleitung

Auf dem lokalen Server ubusrv läuft ein Ubuntu 22.04 LTS mit Docker.

Als Docker Services sind installiert:

- **nginx** - ein Website für Tests http://ubusrv:80/
- **Maria DB** - eine Datenbank, die hauptsächlich für Bookstack genutzt wird
- **Mongo DB** - eine Non SQL Datenbank für Tests
- **Oracle XE DB** - eine Oracle 21g Datenbank für Tests
- **Bookstack** - ein Dokumentationswerkzeug http://ubusrv:9000/
- **ConfigServer** - ein beispielhafter Spring Boot Config Server
- **Adminer** - ein minimaler phpMyadmin clone http://ubusrv:8080/
- **Mongo Express** - eine Bedienoberfläche für Mongo http://ubusrv:8081/

Die Konfiguration erfolgt in der Docker Compose Standarddatei docker-compose.yml im Verzeichnis ~/workspace/docker.

Services: docker_compose.yml

ubusrv stellt diverse Services wie Bokkstack, Oracle-XE Datenbank, Maria-DB etc. bereit. Alle Services werden über Docker Images realisiert. Die Images und deren Parameterisierung werden in einer einzigen docker-compose.yml Datei verwaltet.

```
#
# Start with sudo docker-compose up -d
# Stop  with sudo docker-compose down
#
version: '3.3'

#
# declare Services
#
services:
  #
  # nginx
  #
  nginx:
    image: nginx
    ports:
      - 80:80
      - 443:443
    volumes:
      - /home/ralf/workspace/docker/data/nginx:/usr/share/nginx/html
    restart: always

  #
  # Maria DB Database
  #
  mariadb:
    image: mariadb
```

```
  volumes:
    - /home/ralf/docker/data/mariadb:/var/lib/mysql
  restart: always
  ports:
    - 3306:3306
  environment:
    MYSQL_ROOT_PASSWORD: bistef03

#
# Mongo DB
#
mongo:
  image: mongo
  volumes:
    - /home/ralf/docker/data/mongo:/data/db
  restart: always
  ports:
    - 27017:27017
  environment:
      MONGO_INITDB_ROOT_USERNAME: ralf
      MONGO_INITDB_ROOT_PASSWORD: bistef03

#
# Oracle XE DB
#
oracle:
  image: gvenzl/oracle-xe:full
  volumes:
    - /home/ralf/docker/data/oracle:/data/oradb
  restart: always
  ports:
    - 1521:1521
  environment:
    ORACLE_PASSWORD: bistef03
    APP_USER: anlei
    APP_USER_PASSWORD: anlei

#
# Bookstack
#
```

```
  bookstack:
    image: lscr.io/linuxserver/bookstack
    container_name: bookstack
    environment:
      - PUID=1000
      - PGID=1000
      - APP_URL=http://ubusrv:9000
      - DB_HOST=db
      - DB_USER=root
      - DB_PASS=bistef03
      - DB_DATABASE=bookstack
    volumes:
      - /home/ralf/docker/data/bookstack:/config
    ports:
      - 9000:80
    restart: unless-stopped
    depends_on:
      - mariadb

  #
  # Spring Boot Config Server
  #
  config:
    depends_on:
      - mariadb
    image: hyness/spring-cloud-config-server
    restart: always
    ports:
      - 8888:8888
    environment:
      SPRING_PROFILES_ACTIVE: jdbc
      SPRING_DATASOURCE_URL: jdbc:mariadb://db:3306/cloud_config
      SPRING_DATASOURCE_USERNAME: root
      SPRING_DATASOURCE_PASSWORD: bistef03
      SPRING_CLOUD_CONFIG_SERVER_JDBC_SQL: 'SELECT `key`, value FROM PROPERTIES WHERE
application=? and profile=? and label=?'

  #
  # Adminer: Small implementation of phpMyAdmin
  #
```

```
  adminer:
    depends_on:
      - mariadb
    image: adminer
    restart: always
    ports:
      - 8080:8080

  #
  # Mongo-Express
  #
  mongo-express:
    depends_on:
      - mongo
    image: mongo-express
    restart: always
    ports:
      - 8081:8081
    environment:
      ME_CONFIG_MONGODB_ADMINUSERNAME: ralf
      ME_CONFIG_MONGODB_ADMINPASSWORD: bistef03
      ME_CONFIG_BASICAUTH_USERNAME: ralf
      ME_CONFIG_BASICAUTH_PASSWORD: bistef03
```

Oracle XE DB Installation

> Die Oracle XE Datenbank läuft nur auf Intel-Prozessoren. Auf ARM basierten Servern wie z. B. dem Raspberry PI lässt sich die Datenbank nicht installieren.

Um mit der freien Oracle XE Datenbank arbeiten zu können, wird diese auf dem lokalen Server ubusrv als Docker Container installiert. Hierzu ist die Datei ~/workspace/docker/docker-compose.yml um folgenden Eintrag zu ergänzen:

```
    #
    # Oracle XE DB
    #
    oracle:
        image: gvenzl/oracle-xe:full
        volumes:
            - oracle_data:/data/oradb
        restart: always
        ports:
            - 1521:1521
        environment:
            ORACLE_PASSWORD: bistef03
            APP_USER: anlei
            APP_USER_PASSWORD: anlei


...
#
# Declare Volumes
#
volumes:
    db_data: {}
    mongo_data: {}
    oracle_data: {}
```

Nach einem `sudo docker-compose down` gefolgt von einem sudo `Docker-compose up &` wird das Image vom Docker-Hub heruntergeladen, installiert und gestartet.

Die Datenbank ist nun über das Port 1521 erreichbar. Neben dem Datenbank Basis-Container XE-DB wurde im Datenbank-Container XEPDB1 automatisch das Schema ANLEI angelegt. Auslöser sind die Parameter APP_USER und APP_USER_PASSWORD in obiger .yml Datei. Somit können zwei DBA-Verbindungen und eine USER-Verbindung zur Datenbank aufgebaut werden. Die Verbindungsparameter für diese lauten für den SQL-Developer wie folgt:

```
SYS@XE-DB
=========
□Datenbanktyp: □□□Oracle
□Authentifizierungstyp: □Standard
□Benutzername:□□□sys
□Rolle:□□□□□SYSDBA
□Kennwort:□□□□bistef03
□Verbindungstyp:□□□Einfach
    Hostname:□□□□ubusrv
    Port:□□□□□1521
    SID:□□□□□XE
    Servica-Name:□□□(kein Eintrag)

SYS@XEPDB1
==========
□Datenbanktyp: □□□Oracle
□Authentifizierungstyp: □Standard
□Benutzername:□□□sys
□Rolle:□□□□□SYSDBA
□Kennwort:□□□□bistef03
□Verbindungstyp:□□□Einfach
    Hostname:□□□□ubusrv
    Port:□□□□□1521
    SID:□□□□□(kein Eintrag)
    Servica-Name:□□□XEPDB1

ANLEI@XEPDB1
==========
□Datenbanktyp: □□□Oracle
□Authentifizierungstyp: □Standard
□Benutzername:□□□anlei
□Rolle:□□□□□Standard
□Kennwort:□□□□anlei
□Verbindungstyp:□□□Einfach
```

```
    Hostname:□□□□ubusrv
    Port:□□□□□1521
    SID:□□□□□(kein Eintrag)
    Servica-Name:□□□XEPDB1
```

Um ein weitere Schemas im Arbeitscontainer XEPDB1 anzulegen, sind nachfolgende SQL-Befehle als SYS-User auszuführen:

```
//
// Es wird davon ausgegangen, dass SYS sich auf dem Basis-Container
// angemeldet hat. Sollte sich SYS auf dem dem Arbeits-Container
// XEPDB1 angemeldet haben, können die beiden ALTER SESSION Befehle
// zu Beginn und zum Ende des Skripts auskommentiert werden.
//
// Der Term <schemaname> ist durch den gewünschten Schema-Namen zu
// ersetzen.
//
ALTER SESSION SET CONTAINER = XEPDB1;

CREATE BIGFILE TABLESPACE
    <schemaname>_bfts
    DATAFILE '<schema_name>_bfts1.dbf'
    SIZE 20M
    AUTOEXTEND ON
;

CREATE USER
    <schemaname>
    IDENTIFIED BY <password>
    DEFAULT TABLESPACE <schemaname>_bfts
    QUOTA UNLIMITED ON <schema_name>_bfts
;

GRANT ALL PRIVILEGES TO <schema_name> WITH ADMIN OPTION;

ALTER SESSION SET CONTAINER = CDB$ROOT;
```

Oracle XE DB: DBMS_CRYPTO

Um Passwörter sicher in eigenen Tabellen zu speichern, sollte stets eine sichere HASH-Funktion verwendet werden. Oracle bietet eine solche im Package SYS.DBMS_CRYPTO an.

Das Package einem User nutzbar machen

Der Zugriff auf das Package für "normale" User kann erst nach einem GRANT EXECUTE erfolgen. Hierzu ist wie folgt vorzugehen:

- als User SYS auf dem Arbeitsmarkt-Container XEPDB1 anmelden
- den Befehl `grant EXECUTE on sys.dbms_crypto to <user>;` ausführen

Password-Hash

Kennwörter sollten immer als nicht rückrechenbare Hash-Werte gespeichert werden. Sie müssen nicht entschlüsselt werden, um sie auf Gleichheit zu überprüfen. Stattdessen überprüft man den Hash-Wert auf Gleichheit. Hier bietet das Package eine sinnvolle Unterstützung, da es Strings in einen nach derzeitigem Stand sicheren Hash-Wert umwandelt. Allerdings ist darauf zu achten, dass nur sichere Algorithmen verwendet werden, denn das Package bietet aus Gründen der Kompatibilität auch unsichere Algorithmen wie MD5 an. Um dieses Problem zu umgehen und als eine für eine Datenbankunabhängige Nutzung sinnvolle Erleichterung sollte eine Funktion installiert werden, die den Aufruf des Package abstrahiert. Für Oracle sieht diese Funktion wie folgt aus:

```
//
// Funktion zum Erzeugen eines sicheren HASH aus dem übergenen String
// Es wird der SHA-512 Algorythmus verwendet (gekennzeichnet durch den
// 2. Parameter im dbms-Aufruf). Er ist der derzeit sicherste angebotene
// Algorythmus.
//
CREATE OR REPLACE
FUNCTION CRYPTO_HASH (v_input VARCHAR2)
RETURN RAW DETERMINISTIC
AS
   PRAGMA UDF;
BEGIN
```

```
   RETURN dbms_crypto.hash(utl_raw.cast_to_raw(v_input), 6);
END CRYPTO_HASH ;
/
```

Durch den Aufruf `select crypto_hash('Das wäre mal ein Kennwörtlein') as password from dual;` erhält man als HASH-Wert etwa folgenden 128 Zeichen langen String: `343D52872991F63A64191971501B5515703A34357E2A54956D3AFF79C225AC45BF9DF4CC7EC497211CE75FCE51A2F0A1E621DEA3E656480C5E55ABFF61D8BA6F`.

Decrypt / Encrypt

Die Funktionen zum Verkauf- und Entschlüsseln eines Textes sollten nur mit äußerster Vorsicht verwendet werden. Wenn sich ein Angreifer Zugriff auf die Datenbank verschaffen konnte, kann er auch mit Hilfe der Standardfunktionen derart verschlüsselte Texte entschlüsseln. Sie eignen sich daher nicht für Kennwörter, die wieder entschlüsselt werden müssen (z. B. für Geschütze API-Zugriffe). Hier bietet Vault eine deutlich sicherere Lösung.

Oracle XE DB: Mac M1 Docker Image

Um die Oracle Datenbank Oracle-XE auf einem Apple-Rechner mit M1 Chip zu installieren, wird die Hilfsoftware COLIMA benötigt. Sie macht Docker-Images, die für die X86/AMD-Architektur erstellt wurden unter der ARM-Architektur des Apple Silicon lauffähig. Installiert wird sie mit Homebew:

```
brew install colima
```

Anschließend kann, wenn noch nicht geschehen, der Docker Client ebenfalls mit Hilfe von Homebrew installiert werden:

```
brew install docker
```

Sind beide Pakete erfolgreich installiert, muss zuerst Colima gestartet werden. Dazu wird über Kommandozeilenparameter die erforderliche Architektur eingestellt. Mit weiteren Parametern kann Einfluss auf die Zahl der CPUs, die Größe des RAMs und die Größe des Dateisystems Einfluss genommen werden. Genauere Informationen zu den Parametern liefert die Hilfe. Der einfache Start für die Oracle-XE sieht wie folgt aus:

```
#
# For help use: colina start --help
#
colina start --arch x86_64 --memory 4
```

Die VM für Docker Container wird nun gestartet und es dauert eine Weile, bis das Terminal wieder Befehle entgegennimmt. Ist der Eingabeprompt wieder vorhanden, kann die Datenbank gestartet werden:

```
docker run -d -p 1521:1521 -e ORACLE_PASSWORD=geheimesPassword gvenzl/oracle-xe
```

Zum Stoppen der Datenbank erst das Image wie gewohnt anhalten, dann Colima stoppen: `colima stop`.

Spring Boot

Spring Boot: Docker Image

Dieser Artikel beschreibt am Beispiel des Spring-Boot Erueka Discovery Server, wie ein Spring Boot Server in ein Docker Image verpackt und gestartet wird. Vorausgesetzt wird eine Java 1.8 Installation, eine betriebsbereite Docker Installation und eine Java IDE bzw. ein guter Editor. Ein Build-Tool wie Maven oder Gradle ist entweder Bestandteil der IDE oder installiert.
Die Beispiele wurden mit der Docker Engine 20.10.5, der Eclipse-IDE 2021-03 und dem Java 1.8.0_271 auf einem MacBook Air erstellt. Die Eclipse-IDE wurde um die Module Eclipse-Docker-Tooling 5.2 und Spring-Tools 4 aus dem Eclipse Marketplace erweitert.

Das Beispiel nimmt keine Rücksicht auf Sicherheitsaspekte wie z. B. Spring Boot Security, sichere Protokolle wie ssl, OAuth2 etc. Es soll nur beispielhaft gezeigt werden, welche grundsätzlichen Schritte notwendig sind, um einen Spring Boot Server in ein Docker Image zu packen.

Anlegen eines Spring-Boot Projektes

Ein Spring Boot Projekt mit dem Eureka Discovery Server wird über *Projekte/Spring Boot/Spring Starter Project* erzeugt. Im Projektdialog legt man auf der ersten Seite Projektname und Group, Artefact-ID und weiter Grundeinstellungen für das Projekt fest. Der Projekttyp ist Maven, das Packaging jar, die Java-Version 8 und die Sprache Java.

Auf der zweiten Seite des Dialogs werden der Eureka Server und Spring Boot Actuator ausgewählt. Sie können bequem über die Suchfunktion durch Eingabe von Eureka und Actuator gefunden werden. Mit Finish wird anschließend das Projekt erstellt.

Wer nicht über eine entsprechende Erweiterung in seiner IDE verfügt, kann alternativ den Spring Initializer auf https://start.spring.io verwenden. Die Vorgehensweise ist ganz ähnlich. Die erzeugte ZIP-Datei enthält die benötigte Projektstruktur.

Anpassungen im Projekt

Um den Eureka-Server zu aktivieren ist die Main-Class um die Annotation @EnableEurekaServer zu erweitern. Die Annotation muss zusätzlich importiert werden:

```
package de.rastef.jusoko.esrserver;


import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class JusokoEsrserverApplication {

    public static void main(String[] args) {
        SpringApplication.run(JusokoEsrserverApplication.class, args);
    }
```

Zusätzlich sind in den Application Properties neue Properties einzutragen:

```
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
management.endpoints.web.exposure.include=info,health,env
```

In der ersten Zeile weisen wir dem Server ein Port zu. Über http://localhost:8761/ ist nach dem Start des Servers dessen Dashboard erreichbar. Die zwei folgenden Zeilen, verhindern, dass sich der Server selbst bei anderen Eureka-Servern zu registrieren versucht. Minder letzten Zeile werden in der Actuator-Schnittstelle die Funktionen info, health und env für das Webinterface freigegeben. Standardmäßig sind nur die ersten beiden Funktionen über das Web erreichbar. Eine Actuatorfunktion wird über die URL http//localhost:8761/actuator/<function> aufgerufen. Für die Health-Funktion lautet der Aufruf also http://localhost:8761/actuator/health.

Nach diesen Änderungen kann das Projekt mit maven clean und maven install übersetzt werden. Ein erster Start erfolgt in Eclipse mit *Run As Spring Boot App*. Die zuvor genannten URLs sollten nun erreichbar sein.

Dockerize it

Um den Server in einen Docker-Container zu packen, muss die Datei Dockerfile im Rootverzeichnis des Projektes angelegt werden. Die Datei erhält folgenden Inhalt:

```
FROM openjdk:8-jdk-alpine
RUN addgroup -S esrsrvr && adduser -S esrsrvr -G esrsrvr
USER esrsrvr:esrsrvr
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Das Image benötig eine Java Laufzeitumgebung. Für den Bau des Image wird das schlanke Alpine-JDK genutzt. Da der eigentliche Server nicht als root laufen soll, wird eine Gruppe esrsrvr und ein User esrsrvr angelegt, der der Gruppe esrsrvr zugewiesen wird. Unter diesem User und dieser Gruppe wird dann der Server gestartet. Dazu wird die durch den Build erzeugte Datei target/app.jar in das Image kopiert und der entsprechende Entrypoint gesetzt. Mit

```
docker build -t rastef/esr-service .
```

wir nun das Image gebaut. In diesem Fall heißt das Image rastef/esr-service.

Docker Image starten

Das erste mal wird das Image mit dem Befehl

```
docker run -p 8761:8761 -e SPRING_PROFILES_ACTIVE=prod,actuator --name esr rastef/esr-service
```

gestartet. Der von Docker erzeugte Container erhält dabei den Namen esr. Über diesen ist er bei späteren Start-/Stop-Kommandos einfacher anzusprechen. Die URLs aus dem obigen Test sollten nun genauso funktionieren. Angehalten wird der Container mit STRG+C.

Der nächste Start desselben Container kann mit

```
docker start esr
```

erfolgen. Angehalten wird er mit

```
docker stop esr
```

Wird der Container nicht mehr benötigt, sollte er mit

```
docker rm esr
```

entfernt werden.

Spring Boot: SQL für Config Server

Spring Cloud Config Server Properties Tabelle für MySQL / MariaDB:

```
SET NAMES utf8;
SET time_zone = '+00:00';
SET foreign_key_checks = 0;
SET sql_mode = 'NO_AUTO_VALUE_ON_ZERO';

DROP DATABASE IF EXISTS `cloud_config`;
CREATE DATABASE `cloud_config` /*!40100 DEFAULT CHARACTER SET utf8 */;
USE `cloud_config`;

DROP TABLE IF EXISTS `PROPERTIES`;
CREATE TABLE `PROPERTIES` (
  `application` varchar(255) NOT NULL,
  `profile` varchar(255) NOT NULL,
  `label` varchar(255) NOT NULL,
  `key` varchar(255) NOT NULL,
  `value` varchar(255) NOT NULL,
  PRIMARY KEY (`application`,`profile`,`label`,`key`),
  KEY `application_profile_label` (`application`,`profile`,`label`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `PROPERTIES` (`application`, `profile`, `label`, `key`, `value`) VALUES
('cfs', 'dev', '0.0.0.1', 'jusoko.cfs.long_name', 'Spring Boot Configuration Server'),
('cfs', 'dev', '0.0.0.1', 'jusoko.cfs.info', 'CFS 0.0.0 Build 1'),
('cfs', 'dev', '0.0.0.1', 'jusoko.cfs.active_since', '2021-04-18'),
('cfs', 'dev', '0.0.0.1', 'jusoko.cfs.short_name', 'CFS');
```

Kubernetes

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
```

Kubernetes

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 Eir
>
> 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

Kubernetes

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

Kubernetes

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

Kubernetes

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

Kubernetes

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.

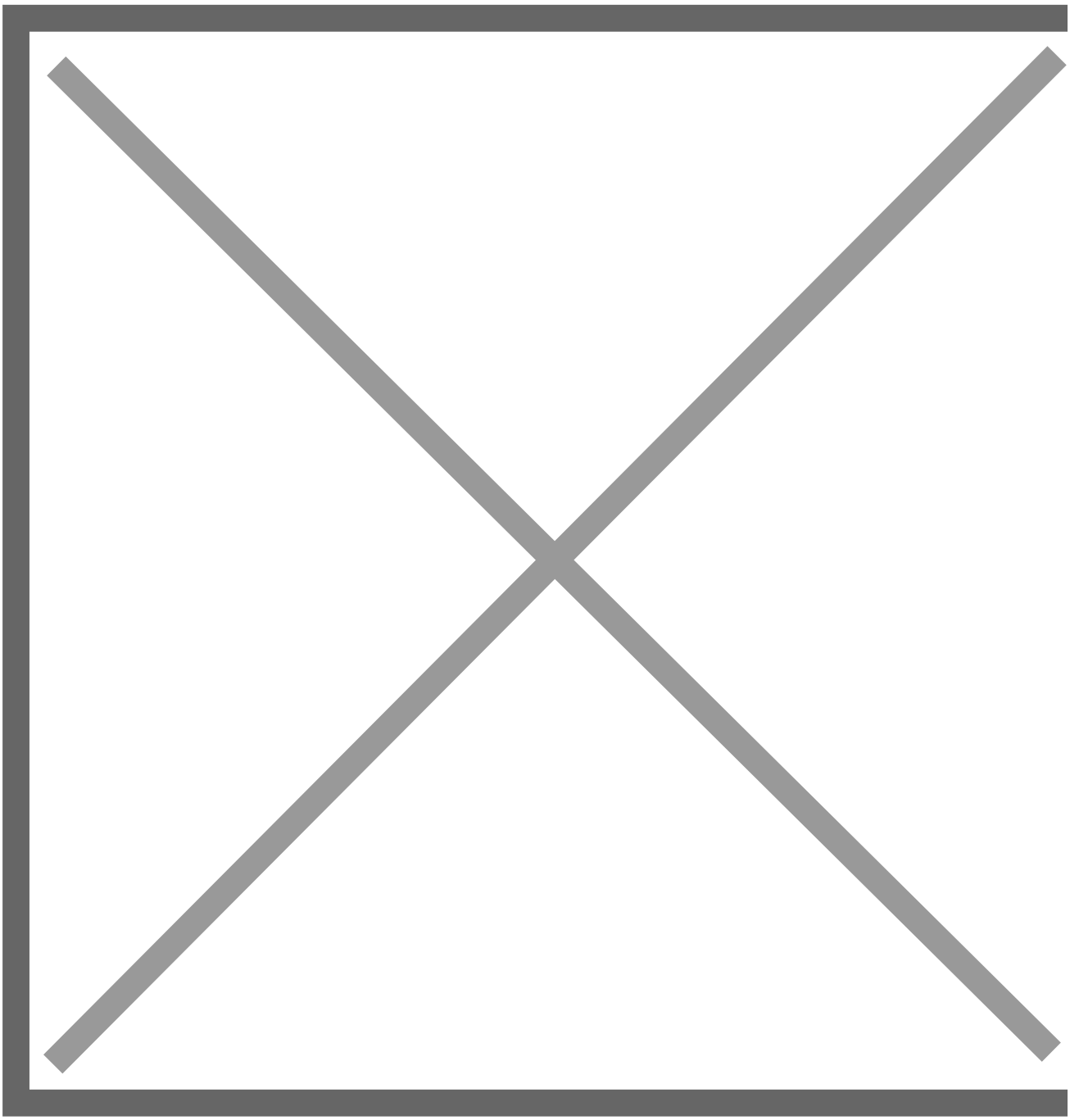
Von Jan Mahn

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

Raspberry PI 3 mit Docker

Docker installieren

Wie vor jeder Installation sollte das System auf den neuesten Stand gebracht werden. Dazu ist der zur genüge bekannte Zweiklang auszuführen:

```
sudo apt update && sudo apt upgrade
```

Sollte Docker schon installiert sein, ist es vor der erneuten Installation zu entfernen:

```
sudo apt remove docker-ce*
sudo apt-get autoremove
```

Die Installation der aktuellen Docker Engine übernimmt ein Skript von Docker:

```
curl -sSL https://get.docker.com | sh
```

Ob Docker installiert wurde und läuft, wird mit folgenden Befehlen geprüft:

```
sudo systemctl status docker
sudo docker ps
```

Damit der aktive Useraccount die docker Befehle nicht mit sudo ausführen muss, kann ihm die Gruppe docker zugewiesen werden. Aber Achtung: Der Account erhält hinsichtlich docker root Rechte!

```
sudo usermod -a -G docker $(whoami)
```

Damit die Rechte wirksam werden, bitte einmal abmelden und wieder anmelden.

> In vielen Anleitungen wird empfohlen docker-compose via python3:pip3 zu installieren. In der docker Version, die Grundlage für diesen Artikel ist, ist compose mittlerweile ein docker command. Auf die Python3 Version kann also verzichtet werden.

Automatisierung mit Ansible

Telerec't

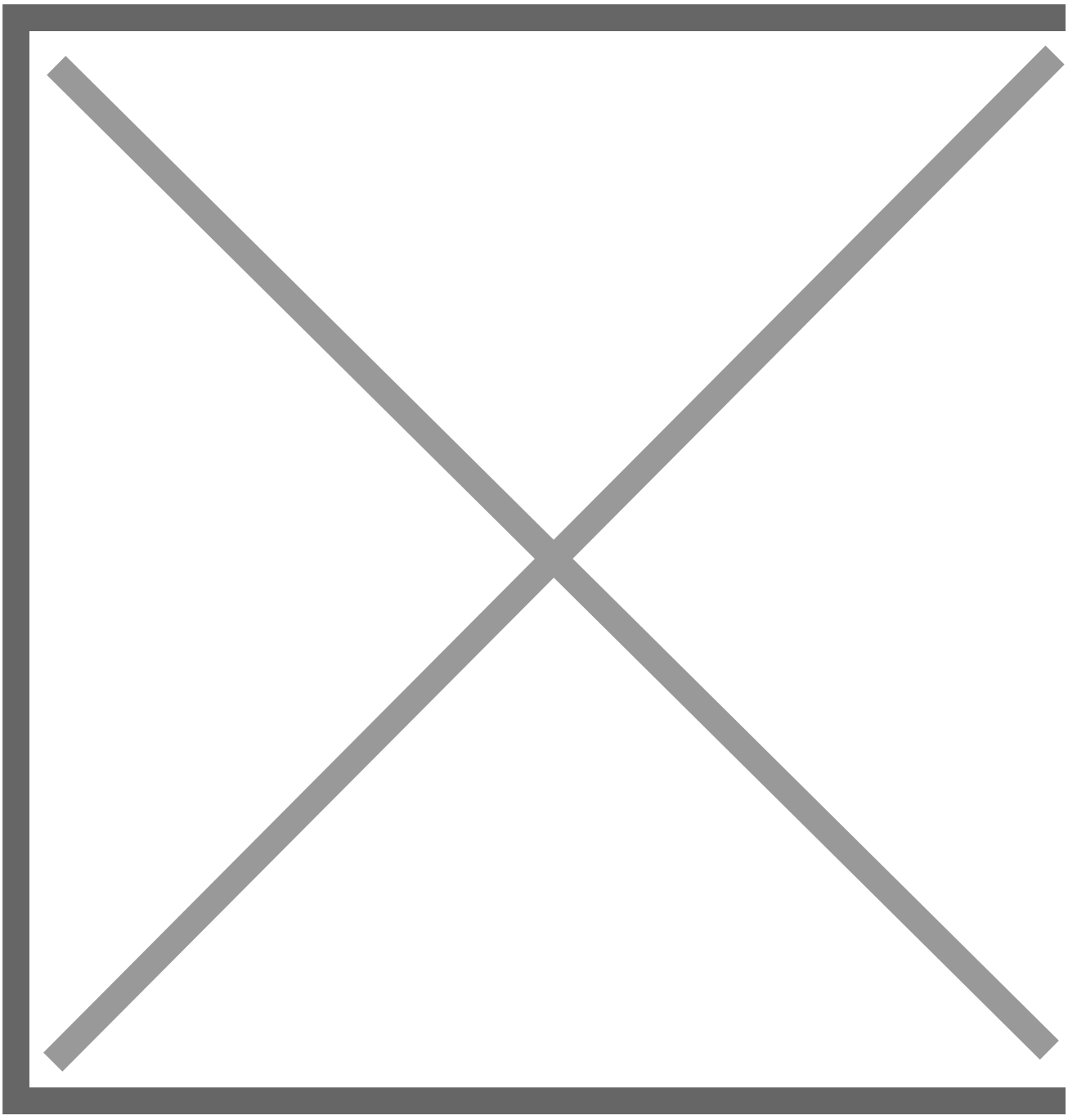


Bild: Thorsten Hübner

Server im Rechenzentrum oder daheim mit Ansible automatisieren

Ansible ist die Automatisierung der Admin-Tätigkeit: Installieren, Kopieren, Patchen – Ansible setzt Ihre Befehle reproduzierbar um, auf so vielen Servern, wie Sie wollen. Gleichzeitig sind die Anweisungen eine vollständige Dokumentation. Als Beispiel bauen wir einen Server-Baukasten namens „Telerec't".

Von Klaus Greff und Pina Merkert

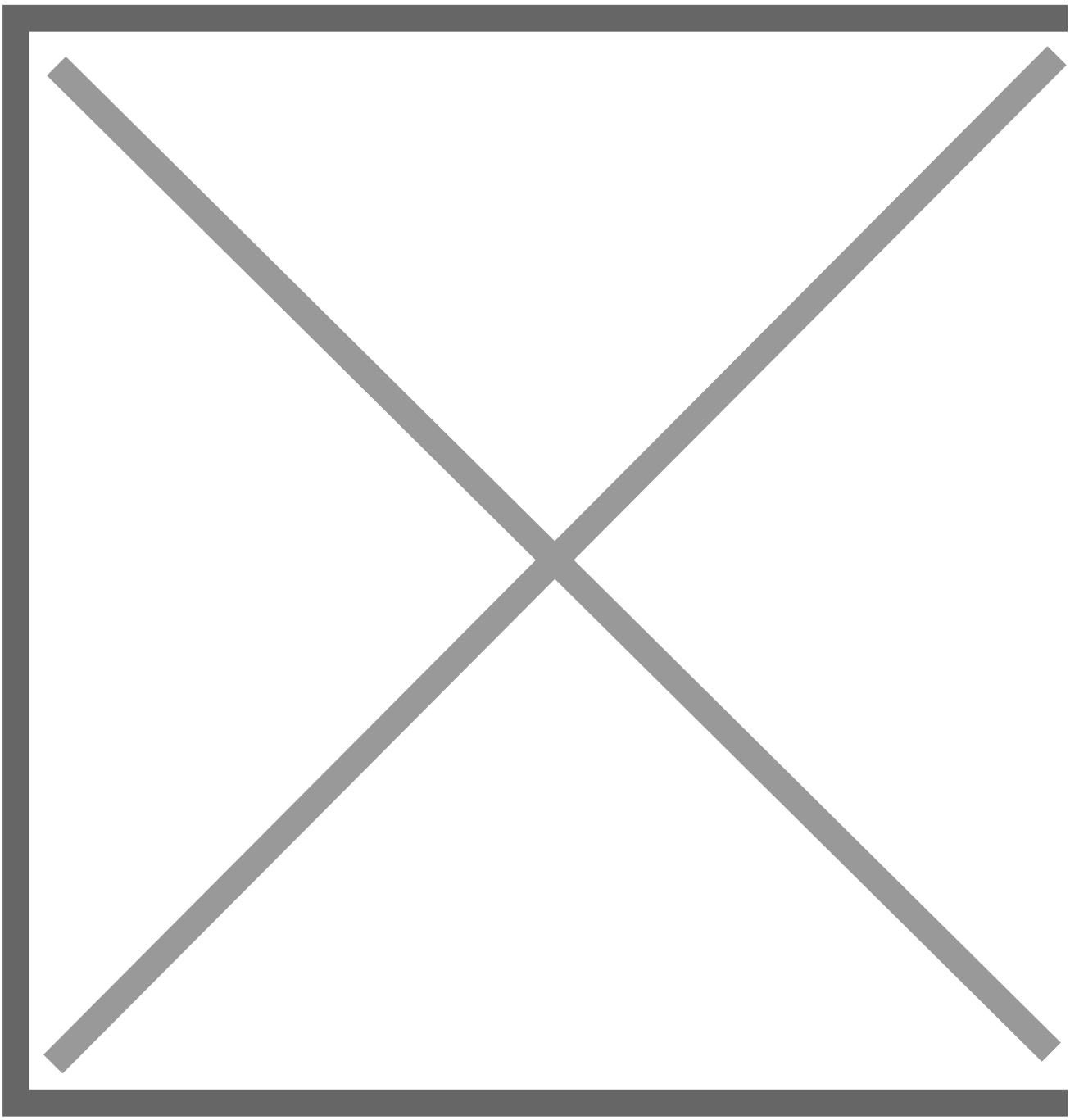
kompakt

- Ansible richtet vollautomatisch Server ein. Was das Programm tun soll, steht in sogenannten Roles und Playbooks.
- Infrastructure-as-Code: Die Roles und Playbooks dokumentieren alle Arbeitsschritte lückenlos und Sie können Programmiererwerkzeuge wie Ihre Lieblings-IDE und Git für die Administration nutzen.
- Als Beispiel haben wir „Telerec't", ein Server-Setup im Baukastensystem, zusammengestellt. Wir werden es in folgenden Artikeln erweitern.

Login per SSH, Docker installieren, immer die gleichen Container hochfahren. Die Arbeit eines Admins kann eintönig sein. Nur nicht vertippen! Wenn Sie da keine Lust drauf haben, sind Sie nicht allein.

Da man ohnehin Computer benutzt, um Computer zu administrieren, lässt sich das alles automatisieren. Eine beliebte Software dafür ist Ansible. Stellen Sie sich das Programm wie einen etwas unselbstständigen Admin-Kollegen im Homeoffice vor. Weil der ein Roboter ist, können sie nicht normal mit ihm reden, sondern geben ihm stattdessen sogenannte „Playbooks". Das sind To-do-Listen, die er stets fehlerfrei und in Rekordzeit abarbeitet.

Im Grunde verschiebt Ansible Know-how vom Notizzettel des Admins in ausführbare Playbooks. Der Fachbegriff dafür lautet „Infrastructure-as-Code". Playbooks lassen sich im Unterschied zu einer Zettelwirtschaft mit Git verwalten und mit einem vertrauten Texteditor oder einer verlässlichen Entwicklungsumgebung für Programmierer editieren.



Playbooks enthalten eine geordnete Liste an Teilaufgaben („Tasks“), die Robo-Kollege Ansible abzuarbeiten hat. Für eine bessere Übersicht können Sie zusammengehörende Aufgaben in „Roles“ organisieren. Eine Menge Fehler beim Administrieren von Servern entstehen durch Copy & Paste und danach vergessene Anpassungen. Damit Sie gar nichts anpassen müssen, unterstützt Ansible Variablen und Templates.

Letztendlich passiert aber nichts anderes als ein Log-in auf dem Server mit der Secure-Shell SSH [1]. Auf dem müssen Sie also keine Client-Software installieren. Ansible führt auf dem Server nur stinknormale Shell-Skripte aus, die Sie nicht mal selbst schreiben müssen. Bei all der Admin-Magie, die wir Ihnen im Folgenden vorführen, sollten Sie nicht vergessen, dass Sie alles auch per Hand ausführen könnten. Wenn Sie und kein anderer Admin eine Idee haben, wie es per Hand geht, werden Sie es Ansible auch nicht beibringen können.

Als Beispiel automatisieren wir die Administration eines eigenen Heim- und eines Root-Servers mit Debian oder Ubuntu und unserem Baukasten „Telerec't“ (von „tele erect“ – „aus der Ferne aufgerichtet“). Der Heimserver soll später mal Smart-Home-Dienste wie Mosquitto (MQTT-Broker) und NodeRed (No-Code Automatisierungsregeln) ausführen, der angemietete Root-Server im Rechenzentrum Wordpress (Blog-Software) und Nextcloud (Dateisynchronisation) beherbergen. SSH-Schlüssel hinterlegen sowie Basis-Pakete und Docker installieren ist für beide Server gleich. Im nächsten Artikel zu Ansible stellen wir vier Dienste vor, die in unseren Augen auf jedem Server laufen sollten. Erst danach folgen in weiteren c't-Ausgaben die eigentlichen Bausteine in Form zusätzlicher Ansible-Roles als eigene Artikel, sodass Sie ganz individuell auswählen können, was Sie auf Ihrem Server installieren wollen.

Die Bausteine setzen wir als Git-Submodules um. Submodules sind ein eher selten genutztes Feature der Versionsverwaltung Git [2, 3]. Jedes Submodule ist ein simpler Unterordner, aber gleichzeitig auch ein eigenständiges Git-Repository. Das binden Sie so in ein Basis-Repository ein, dass dieses protokolliert, auf welchem Stand das Gesamtkonstrukt war. Zum Baukasten wird das, weil Sie die Submodules unverändert in Ihr Basis-Repository einbinden können. Sie pflegen nur dieses eine Repository, denn das bildet Ihr Setup vollständig ab. Updates in den Submodules spielen Sie aber trotzdem mit nur einem Befehl ein.

Infrastructure-as-Code

In diesem Artikel zeigen wir, wie Sie Ansible installieren und eigene Playbooks schreiben. Im ersten Playbook dieses Artikels übertragen wir kryptografische Schlüssel, damit Sie und Ansible sich danach mit der Secure Shell SSH [1] ohne Passwort am Server anmelden können. Außerdem werden wir Docker und `docker-compose` installieren. Docker ist meist die erste Wahl, um containerisierte Anwendungen auszuführen. In Container verpackte Anwendungen kommen sich nicht gegenseitig in die Quere. Fertige Images lädt Docker aus dem Docker Hub herunter. Falls Ihnen eine große Auswahl fertiger Images nicht wichtig ist, können sie Telerec't stattdessen auch mit der Docker-Alternative Podman nachbauen. Wir werden für Telerec't aber Docker benutzen.

Ein Wort der Warnung: Telerec't ist entstanden, weil wir ein Setup für unsere eigenen Server gebraucht haben. Wir veröffentlichen das Basis-Repository und mehrere Submodules, weil diese ein hervorragendes Beispiel sind, wie Sie Ansible für Ihren Server nutzen können. Denken Sie dabei aber bitte mit, prüfen Sie, was wir machen, und tragen Sie gern Verbesserungsvorschläge als Pull-Requests an uns heran. Telerec't ist nämlich mit all seinen Bausteinen Open Source und unsere Konfiguration haben wir unter der Lizenz GPLv3 veröffentlicht. Wir leisten keinen professionellen Support und geben keine Garantie, die Repositories in Zukunft zu pflegen. Sehen Sie das Projekt als Freie Software ohne Maintainer. Forken Sie die Repositories, falls Sie Maintainer werden wollen.

Hardwareanforderungen

Mit Telerec't haben wir sowohl einen Heimserver als auch einen virtualisierten Server im Rechenzentrum aufgesetzt. Wir empfehlen mindestens 8 GByte RAM und zumindest zwei CPU-

Kerne. Mit einem Raspi Zero würden Sie sich keinen Gefallen tun, der große Raspi 5 (8 GByte) mit einer NVMe-SSD käme als kleiner Heimserver aber infrage. Zwei offizielle M.2-Adapter für den Raspi 5 sind für Frühjahr 2024 angekündigt. Momentan würden wir einen Heimserver mit kleinem Mainboard und einem sparsamen x68-Prozessor empfehlen (beispielsweise ein Billig-Barebone aus [5]).

In Rechenzentren ist die kleine Hardware ein virtualisierter Anteil an einem größeren Server. Angebote ab 15 Euro pro Monat sind meist stark genug für ein Dutzend Webserver-Container, sofern nicht viele Anfragen kommen. Hier steigen die Kosten, falls Sie viel Speicherplatz für eine Nextcloud brauchen. Wir mieten beispielsweise einen Server mit 16 virtuellen Kernen, 24 GByte RAM und 500 GByte Speicherplatz für knapp 18 Euro im Monat. Der reicht für einen Mailserver parallel mit vier Wordpress-Containern, einer Nextcloud, Wekan, Vaultwarden und drei eigenen Django-Projekten.

Ansible installieren

Für einen stressfreien Start bestellen Sie einen Root-Server mit einem vorinstallierten Debian oder Ubuntu. Falls Sie einen Heimserver aufsetzen, installieren Sie eines der Systeme wie gewohnt und aktivieren Sie das Log-in per SSH. Bei Raspis empfehlen wir ganz langweilig das Debian-basierte Raspberry Pi OS. Telerec’t funktioniert auf allen Linux-Distributionen mit dem Paketmanager `apt`. Der händische Teil der Installation endet, sobald Sie `ssh`-Zugang haben. Den sollten Sie testen, was Ihren Rechner nebenbei auf dem Server auch in ~/.ssh/known_hosts einträgt (Ansible trägt sich dort nicht automatisch ein).

Danach wechseln Sie zu Ihrem Admin-PC oder -Notebook mit Linux, macOS oder Windows mit dem WSL (Windows Subsystem for Linux). Dort installieren Sie Ansible – es läuft auch nur dort. Der Server bekommt nämlich gar nicht mit, dass Sie sich von Ansible helfen lassen. Sie können auch von mehreren verschiedenen Rechnern aus den Server administrieren. Dafür klonen Sie das Repository mit dem Infrastruktur-Code und geben jedem der Rechner Zugriff auf den Server.

Ansible ist in Python geschrieben, weshalb Sie es auf allen Desktopbetriebssystemen mit Python-Paketmanagern installieren können. Wir empfehlen diese Art der Installation zusammen mit `pipenv`, weil das automatisch ein virtuelles Environment mitverwaltet, sodass unterschiedliche Python-Projekte einander nicht in die Quere kommen können.

Alternativ: In den meisten Linux-Distributionen sind die Ansible-Pakete aus den Paketquellen aktuell genug. Wenn Sie Ansible auf diesem Weg installieren, läuft es ohne virtuelle Umgebung und Sie können `pipenv run` zu Beginn der folgenden Befehle weglassen. Zusätzlich brauchen Sie dann noch `python-passlib`, sparen sich später aber `pipenv install`. Wir haben den Weg über `pipenv` gewählt, weil Sie dann auf allen Betriebssystemen und Distributionen die aktuelle Version installieren.

Am einfachsten administrieren Sie einen Linux-Server von einem Linux-Rechner aus. Installieren Sie die Pakete für `git`, `sshpass` (fürs Einloggen mit Passwort) und `pipenv` beziehungsweise Ansible (falls Sie die Paketverwaltung bevorzugen). Unter Ubuntu geht das beispielsweise mit dieser Zeile:

```
sudo apt install git sshpass pipenv
```

Unter Windows raten wir zum WSL, das Sie ruckzuck installieren, indem Sie in ein cmd- oder PowerShell-Fenster mit Administratorrechten `wsl --install -d ubuntu` eintippen. Danach öffnen Sie Ubuntu über das Startmenü, was die Ersteinrichtung anstößt, die einige Sekunden dauert und während der Sie einen Benutzeraccount anlegen. In der Ubuntu-Konsole haben Sie danach den Ubuntu-Paketmanager `apt` zur Verfügung, um die restlichen Abhängigkeiten zu installieren:

```
sudo apt install python3 git pipenv sshpass
```

Mac-Nutzern empfehlen wir Homebrew, das Sie mit dem folgenden Befehl installieren:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Danach installieren Sie `git` und `pipenv` mit folgendem Befehl:

```
brew install git pipenv
```

Alternativ können Sie Ansible auf dem Mac ebenfalls mit `brew` installieren und `pipenv run` dann ebenfalls weglassen.

Die nächsten Schritte funktionieren auf allen Systemen gleich.

Privates Repository anlegen

Unser Beispiel-Setup Telerec’t fußt auf der Idee, dass Sie sich in einem privaten Git-Repository im Baukastensystem eine Konfiguration für Ihren eigenen Server zusammenbauen. Dafür brauchen Sie zunächst ein leeres privates Repository. Privat sollte das sein, damit Sie dort die Passwörter hinterlegen können. Auch bei kostenlosen GitHub-Accounts können Sie private Repositories erstellen. Falls Sie GitHub nicht vertrauen, können Sie auch mit `git init` ein lokales Repository anlegen und erst später mit `git add remote` ein Repository zum Synchronisieren hinzufügen, beispielsweise auf einem selbst gehosteten Gitea.

Um Ihnen die Arbeit an Ihrem individuellen Setup zu erleichtern, haben wir ein Beispiel-Repository erstellt, bei dem Sie sich bedienen können (siehe ct.de/yu9e). Statt mit Copy & Paste befüllen Sie Ihr Repository daraus schneller mit `git`:

```
git remote add ct-example https://github.com/ct-Open-Source/telerec-t-base
git pull ct-example main
git remote remove ct-example
```

Danach haben Sie eine Basiskonfiguration für Ansible bestehend aus den Dateien ansible.cfg und hosts sowie den Verzeichnissen group_vars/ und roles/. In letzterem befinden sich bereits Git-Submodule, die aber noch nicht geklont wurden. Das holen Sie mit folgendem Befehl nach:

```
git submodule update --init \
  --recursive
```

Weil Sie den länglichen Befehl für Updates der Submodule immer mal wieder brauchen, lohnt es sich, ein Alias für die Kurzform `git supdate` anzulegen:

```
git config alias.supdate 'submodule update --remote --merge'
```

Nun haben Sie zwar `pipenv` und die nötige Verzeichnisstruktur, aber Ansible und Passlib sind noch nicht installiert. Die stehen nämlich nur als Voraussetzung in der Datei Pipfile im Repository. Die Voraussetzungen installieren Sie aus dem Basis-Ordner des Repositorys mit:

```
pipenv install
```

Setup anpassen

Das neu angelegte und mit Beispieldaten befüllte Repository müssen Sie nun an den eigenen Server anpassen. Tragen Sie dafür zunächst in der zweiten Zeile der Datei `hosts` die statische IP-Adresse Ihres Servers oder bei einem Heimserver optional auch dessen Hostname ein. Passend zu diesem Rechner können Sie nun Variablen in host_vars/<ip_oder_hostname>.yml anlegen. Hat Ihr Server beispielsweise die IP-Adresse 192.168.7.140, lautet der Dateiname also host_vars/192.168.7.140.yml.

> Der Screenshot aus unserer IDE zeigt die Struktur des Setups: Die Submodule sind in den Ordr
> Der Screenshot aus unserer IDE zeigt die Struktur des Setups: Die Submodule sind in den Ordner roles/ eingebunden, Playbooks liegen im Wurzelverzeichnis. Die README.md enthält Befehle zum Herauskopieren.

Wie die meisten Konfigurationsdateien für Ansible ist auch diese Datei im YAML-Format, mit dem Sie Variablen als hierarchische Strukturen definieren können. Das folgende Beispiel müssen Sie für Ihren Server passend anpassen:

```
ansible_user: mariamuster
admin:
  name: mariamuster
  key: "{{ lookup('file', '~/.ssh/id_ed25519.pub') }}"
  email: "maria.muster@gmail.com"
locale: de_DE.UTF-8
timezone: Europe/Berlin
docker_dir: "/docker"
```

In der ersten Zeile steht der lokale Benutzername, der Block darunter definiert einen Account mit Systemverwalterrechten auf dem Server. Die `lookup`-Funktion hinter dem Schlüssel `key` sucht lokal, also auf dem Admin-System, im angegebenen Verzeichnis ~/.ssh/ nach dem öffentlichen Schlüssel für passwortloses Login mit `ssh`. Den Pfad und Dateinamen müssen Sie wahrscheinlich an die Dateinamen und Pfade an Ihrem Rechner anpassen. Moderne Schlüssel heißen oft

id_ed25519.pub, ältere oft id_rsa.pub [4].

Mit der E-Mail-Adresse darunter bauen Sie schon für die Dienste vor, die wir im nächsten Artikel installieren. Wichtig ist hier zunächst nur, dass es sich um eine externe Adresse handelt, die nicht vom eigenen Mailserver auf demselben Server bereitgestellt wird.

In der letzten Zeile können Sie das Verzeichnis umstellen, in dem auf dem Server letztlich die Daten aller Dienste landen.

Struktur erklärt

Ein Ansible-Projekt besteht aus „Tasks“, die zu „Roles“ zusammengefasst sein können. Aus diesen Elementen kann man dann Programme schreiben die im Ansible-Jargon „Playbooks“ heißen. Tasks, die immer gemeinsam ausgeführt werden müssen, sollten zusammen in einer Role stehen. Die Roles sind simple Unterverzeichnisse im roles-Ordner. Playbooks können alternativ auch direkt Tasks enthalten. Solche haben wir für Telerec’t aber nicht gebraucht, weil sie in Roles stets besser aufgehoben waren.

> Im roles/-Ordner gibt es Unterverzeichnisse für jeden Dienst. Was in files/ liegt, kann der copy-
> Im roles/-Ordner gibt es Unterverzeichnisse für jeden Dienst. Was in files/ liegt, kann der copy-Task übertragen, in templates/ liegen Jinja2-Templates für den template-Task bereit. Öffentliche Variablen dürfen in einem Order defaults/ stehen. In tasks/main.yml stehen die Arbeitsanweisungen für Ansible.

Das Herzstück jeder Role ist der tasks-Ordner, in dem Ansible nach einer Datei main.yml sucht. Die kann mit `include_tasks:` auch weitere YAML-Dateien einbinden, automatisch ausgeführt werden die aber nicht. In defaults/main.yml kann man jeweils Role-spezifische Variablen definieren. Der Ordner heißt „defaults“, weil im Playbook definierte Variablen die Variablen aus main.yml überschreiben. Ansible bevorzugt stets spezifisch definierte Variablen gegenüber allgemeinen. In files/ stellt man Dateien bereit, die Tasks unverändert auf den Server kopieren (`copy`-Task), in templates/ landen Dateien in der Template-Sprache Jinja2, die Ansible mit dem `template`-Task ausfüllt und überträgt. Neben Variablen können die Templates auch `if`-Abfragen und Schleifen enthalten.

Mit `ansible-galaxy` lassen sich Roles über eine Art Paketverwaltung teilen. Das haben wir im Playbook initial-setup.yml genutzt und installieren Docker mit einer Role von Raspi-YouTuber Jeff Geerling:

```
pipenv run ansible-galaxy role install geerlingguy.docker
```

Eine so geteilte Role ist im Prinzip nichts anderes als ein Verzeichnis unterhalb von roles/. Für Telerec't haben wir einfach Git-Repositories als Git-Submodules ins roles-Verzeichnis geklont. Wir hätten sie genauso auf Ansible-Galaxy veröffentlichen können. Mit diesen beiden Methoden bauen Sie auch später weitere Bausteine in Ihr Ansible-Setup ein: Sie suchen sich beispielsweise ein passendes Repository bei github.com/ct-Open-Source (siehe ct.de/yu9e) wie 2fauth [5] aus und

klonen es mit dem folgenden Befehl, den Sie im Wurzelverzeichnis des Projekts ausführen:

```
git submodule add https://github.com/ct-Open-Source/telerec-t-2fauth roles/twofauth
```

Beim Basis-Setup sind sechs Roles dabei, die in unseren Augen jeder Server braucht. Das nach roles/system geklonte telerec-t-debian überträgt den SSH-Schlüssel für passwortloses Login, fügt den Benutzer der `sudo`-Gruppe hinzu, installiert mit `apt` Pakete wie `vim`, `wget`, `curl` und `rsync` (welche genau, steht in roles/system/vars/main.yml) und legt das Verzeichnis für Docker-Volumes an. Wir haben dafür /docker/ voreingestellt, Sie können es aber über die Variable in host_vars/ auch ändern. In diesem Verzeichnis landen alle Daten, die Sie regelmäßig sichern sollten.

> Ansible schreibt bunt in die Konsole, welche Tasks geklappt haben. Bricht ein Task mit einem F
> Ansible schreibt bunt in die Konsole, welche Tasks geklappt haben. Bricht ein Task mit einem Fehler ab, werden zuvor ausgeführte Änderungen nicht mehr rückgängig gemacht. Deswegen sollten Tasks einen definierten Zustand herstellen, egal was vorher geklappt hat oder nicht.

Docker findet sich nicht in den voreingestellten Paketquellen. Die Role `geerlingguy.docker` richtet zuerst die Paketquelle mitsamt GPG-Schlüsseln ein und installiert dann Docker und das Compose-Plug-in. Da die Role von Ansible-Galaxy kommt, gibt es für sie keinen Unterordner in roles/. Beide Roles stehen aber im Playbook initial-setup.yml:

```
- hosts: server
  become: true
  roles:
    - system
    - geerlingguy.docker
```

Hinter `hosts:` steht der Name des Servers, den Sie in der Datei „hosts“ angegeben haben. `become: true` verschafft dem Benutzer auf dem Server Root-Rechte und hinter `roles:` folgt die Liste der Verzeichnisse in roles/, die Ansible im Zuge des Playbooks abarbeitet oder die über Ansible-Galaxy geladenen Roles ohne Verzeichnis.

Playbooks ausführen

Ansible will sich standardmäßig ohne Passwort anmelden, was noch nicht geht, wenn der öffentliche SSH-Schlüssel noch nicht hinterlegt ist. Das Playbook initial-setup.yml kopiert den Schlüssel auf den Server; um es auszuführen, müssen Sie Ansible aber anweisen, einmalig nach dem SSH-Passwort und dem `sudo`-Passwort zu fragen:

```
pipenv run ansible-playbook initial-setup.yml -i hosts --ask-pass --ask-become-pass
```

Ansible dokumentiert mit farbigen Ausgaben auf der Kommandozeile die Ausführung aller Tasks. Grün deutet an, dass es nichts tun musste, bei Orange war der Task erfolgreich. Fehlermeldungen markiert Ansible rot.

Der Befehl zeigt, wie mächtig Ansible in der Praxis sein kann. Mit obigem Befehl sind Dutzende Pakete installiert und Schlüssel hinterlegt. Und nichts hält Sie davon ab, zusätzlich auch beliebige Skripte auszuführen, Software zu kompilieren oder Daten umherzuschieben. Alles mit nur einem Befehl, und wenn Sie das wünschen, sogar auf Dutzenden Servern gleichzeitig.

Erfahrungsgemäß werden Sie sich schon nach kurzer Zeit Playbooks für alle Routineaufgaben bei der Server-Administration erstellen. Oft ist die Aufgabe dann mit einem einzelnen Befehl erledigt und Sie können sich wieder interessanteren Tätigkeiten zuwenden. Außerdem dokumentieren Roles und Playbooks exakt, was Sie gemacht haben. So protokollieren Sie Ihre Arbeit nicht nur für Kollegen, sondern auch für sich selbst, und stellen sicher, dass Sie auch Monate später noch nachvollziehen können, welche Schritte Sie genau benutzt haben.

Ähnlich wie das Schreiben von Dokumentation verlangt ein Ansible-Setup aber auch Disziplin. Ändern Sie nicht einfach in einer SSH-Session Konfigurationsdateien auf dem Server. Für Telerec’t müssen sie das eigentlich auch gar nicht. Wenn Sie aber trotzdem einen Hotfix brauchen, denken Sie daran, dass vermutlich eines Ihrer Playbooks den Hotfix zu einem ungünstigen Zeitpunkt überschreiben wird. Fügen Sie der betroffenen Ansible-Role deswegen auch gleich Tasks hinzu, die die gleiche Änderung auch automatisch umsetzen können. Das geht beispielsweise mit Tasks, die mit regulären Ausdrücken Textdateien für die geänderte Konfiguration editieren.

Zugegeben: Wir haben einige Stunden gebraucht, bis wir mit Ansible vertraut genug waren, um die gewohnten Wartungsarbeiten am Server vorzunehmen. Inzwischen hat sich dieser Aufwand aber ausgezahlt: Neue Dienste setzen wir mit weniger Tipparbeit auf und sparen bei jedem etwas Zeit. Durch die gute Dokumentation sind wir deutlich konsequenter beim Einrichten der Dienste, was für uns selbst und auch andere die Übersicht verbessert. Vor allem aber ist das Gefühl verschwunden, dass wir einen früheren Schritt vergessen haben könnten und der neueste Befehl alles kaputtmachen könnte. Unser Setup ist in der Versionsverwaltung gesichert und wir können eine funktionierende alte Konfiguration mit einem Checkout und einem Ansible-Durchlauf ganz schnell wieder rekonstruieren. Administrationsaufgaben reproduzierbar ausführen zu können, gibt ein Gefühl der Sicherheit, das die Einarbeitungszeit schon bald rechtfertigt. Zeitersparnisse, weil Sie Befehle nur genau einmal eintippen müssen, und die Möglichkeit, mit anderen Admins Roles auszutauschen, sind der Zuckerguss auf dem Kuchen.

Im nächsten Artikel zu Telerec’t werden wir uns der Aufgabe widmen, vier essenzielle Docker-Container aufzusetzen und zu starten. *(pmk@ct.de)*

1. Literatur

2. Peter Siering, FAQ: SSH - Secure Shell, c’t 03/2018, S. 158
3. Herbert Braun, Unvergessen, Erste Schritte mit dem Versionskontrollsystem Git und mit GitHub, c’t 5/2014, S. 176
4. Jan Mahn und Merlin Schumacher, Arbeiten mit GitHub, c’t 21/2018, S. 158

5. Niklas Dierking, Schlüsselmeister, Sicher und komfortabel arbeiten mit SSH, c’t 21/2022, S. 172
6. Markus Stubbig, Cloudtresor zum Generieren von Einmalpasswörtern, 2FA-Authentifikator selbst gemacht, c’t 27/2023, S. 146
7. Christof Windeck, Desktop-Duell, So schlägt sich der Raspberry Pi 5 als PC-Ersatz, c’t 29/23, S. 102

Repositories: ct.de/yu9e

Ordnung im Königreich

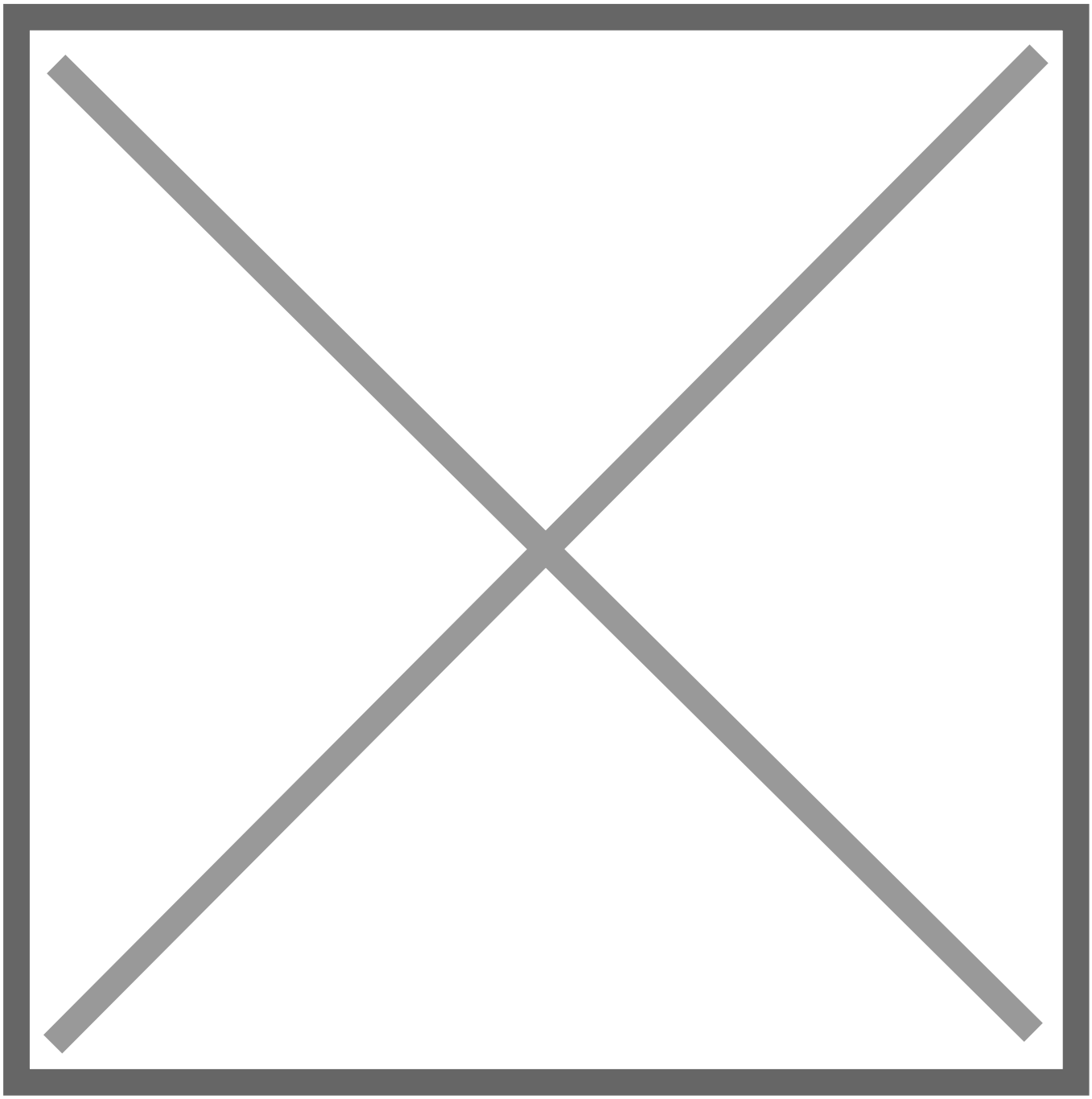


Bild: Thorsten Hübner

Die Grundausstattung für öffentliche Server – automatisiert mit Ansible

Mit einem eigenen Server gewinnen Sie die Datenhoheit zurück. Bevor Sie praktische Dienste installieren können, brauchen Sie aber Zertifikate, einen Reverse-Proxy, automatische Updates und ein Admin-Interface. Mit unserem Server-Baukasten „Telerec’t“ setzen Sie das alles weitgehend automatisch mithilfe von Ansible auf.

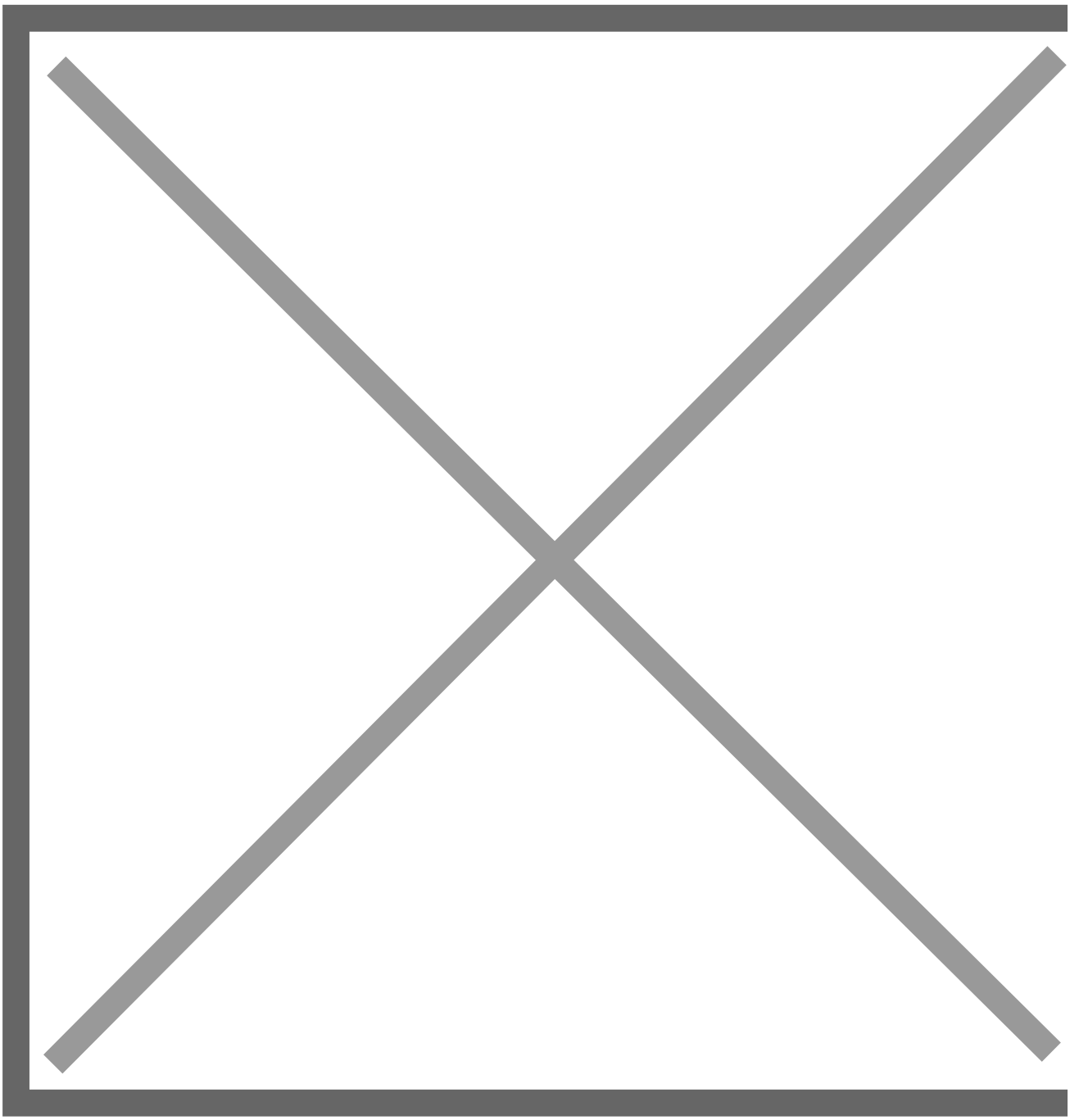
Von Klaus Greff und Pina Merkert
c't 2/2024 Seite 154

c't kompakt

- Ansible richtet vollautomatisch Server ein. Nach der Installation in [1] ergänzen wir in diesem Artikel die Dienste, die jeder im Netz erreichbare Server braucht.
- Heim- und Root- und virtuelle private Server sollten mit signierten Zertifikaten verschlüsselt kommunizieren können, stets aktuell bleiben und leicht administrierbar sein.
- Unser Baukastensystem Telere’ct besteht mindestens aus Docker-Containern für Traefik, Watchtower, Autoheal und Portainer.

Einen Befehl auf der Kommandozeile ausführen, zurücklehnen und zuschauen, wie die eigenen Wünsche umgesetzt werden. Wer seinen Server mit Ansible administriert, kann sich wie ein König fühlen. Wie Sie die ersten Schlachten für die Eroberung Ihres Admin-Reichs schlagen, haben wir in [1] erklärt. Nun gilt es den Server gegen Angriffe abzusichern und einige Helfer für die Verwaltung einzusetzen. Als Monarch lassen Sie nämlich arbeiten und legen nur fest, was Ihre Untertanen zu tun haben.

Den Weg zur Macht beschreiten Sie auf Ihrem Server mit „Teile und Herrsche“. Die Technik dazu heißt „Container“ [2]. Ein Container ist eine auf das Wesentliche zusammengestutzte Linux-Umgebung, die nur einen Dienst ausführt, den aber mit größter Verlässlichkeit. Docker kümmert sich um diese Mini-Linuxe und vernetzt sie virtuell zu einem föderalen Staat. Das ist deutlich effizienter als Virtualisierung, weil alle Container den Kernel des Host mitbenutzen. Der größte Vorteil liegt aber darin, dass Ihr Reich auf den Schultern von Giganten steht: Andere Herrscher teilen bereits optimierte Container-Images über Container-Registries wie den Docker Hub und Sie profitieren von deren Updates.



Regierungsbildung

Damit Ihre Container keinen Staub ansetzen, sollte Ihr Ansible-Reich „Watchtower“ benutzen, das selbst im Container läuft. Es hält Ausschau nach veralteten Docker-Images und ersetzt sie automatisch durch neuere. Dazu gesellt sich „Autoheal“, das regelmäßig prüft, ob alle Container ihre Arbeit verrichten. Wenn nicht, hilft Autoheal ihnen wieder auf die Beine.

Als Außenminister empfehlen wir den Reverse-Proxy „Træfik“ (im Folgenden Traefik geschrieben), der verlässlich alle Anfragen befreundeter Rechner annimmt und an die Container weiterleitet, die dafür zuständig sind. Traefik regelt auch die Transportverschlüsselung der Kommunikation, indem es SSL-Zertifikate von Let’s Encrypt beschafft und automatisch vor Ablauf verlängert.

Vielleicht befürchten Sie, dass es unübersichtlich werden könnte, die vielen Container gleichzeitig im Auge zu behalten. Zum Glück schafft „Portainer“ mit seinem Dashboard Abhilfe. Die Webanwendung listet all Ihre Container auf, erleichtert deren Konfiguration und hilft mit Logs bei der Fehlersuche. Also keine Sorge, dass Sie zu diesem Zweck Ihr Reich zu Fuß bereisen oder mit Konsolenbefehlen hantieren müssten.

Das Repository als Verfassung

Ansible steht Ihnen stets als Protokollchef zur Seite, der den Kleinkram für Sie verwaltet. Wir gehen im Folgenden davon aus, dass Sie Ansible installiert und ein Repository für Ihr Setup angelegt haben. Falls Sie das schnell nachholen wollen, finden Sie alle Infos in [1]. Ein Ansible-Setup besteht immer aus drei Bereichen: Variablen, Roles und Playbooks.

Mit den Variablen legen Sie all die Werte fest, die für Ihr Setup individuell sind. Das sind Benutzernamen, Passwörter, Pfade und Abweichungen von der Standardkonfiguration. Falls eine Variable nur einen bestimmten Server betrifft, legen Sie diese im Ordner host_vars/ in einer YAML-Datei fest, deren Name dem Hostnamen oder der IP-Adresse des Servers entspricht. In unserem in [1] beschriebenen Basis-Setup haben wir so unter anderem den Namen des Standardbenutzers und die Zeitzone des Servers in host_vars/192.168.7.140.yml festgelegt.

> In group_vars/all.yml erstellen Sie einen Block für jeden Dienst, der auch Geheimnisse wie Pas
> In group_vars/all.yml erstellen Sie einen Block für jeden Dienst, der auch Geheimnisse wie Passwörter enthalten darf. Die compose_hull lädt diese Variablen und verbindet Sie mit Standardwerten der Role. Für das Basis-Setup tragen Sie in diese Datei Ihre neu erzeugten Passwörter ein.

Gelten Einstellungen für mehrere Server, sind sie in group_vars/all.yml besser aufgehoben. Dort haben wir Abschnitte für jeden unserer Serverdienste erstellt und dort beispielsweise alle Benutzernamen und Passwörter festgelegt. Da unser Basis-Repository privat ist, können wir damit Passwörter über mehrere Rechner, von denen wir den Server administrieren, synchron halten, beispielsweise der Desktop-PC im Büro und das private Notebook. Falls Ihnen das zu unsicher ist, müssen Sie die Passwortverwaltung in eigene Dienste auslagern. Damit dieser Artikel nicht zu lang wird, haben wir uns gegen diesen Weg entschieden und riskiert, dass GitHub unsere Passwörter kennt.

In unserem Vorlagen-Repository github.com/ct-Open-Source/telerec-t-base enthält die Datei schon Blöcke mit den Variablen der vier Container, die Sie immer brauchen. Die Passwörter müssen Sie aber unbedingt ändern! Der folgende Linux-Befehl erzeugt ein Passwort mit 25 Zeichen und nutzt dafür die besten Zufallszahlen, die der Kernel erzeugen kann:

```
cat /dev/urandom | tr -dc a-zA-Z0-9 | head -c25; echo
```

Nutzen Sie diesen Befehl, um ein `admin_password` für `portainer`, ein `http_token` für `watchtower` und ein `admin`-Passwort für `traefik` zu erzeugen. Letzteres müssen Sie für die HTTP-Basic-Authentifizierung des Webinterfaces von Traefik noch hashen und mit dem Nutzernamen

kombinieren. Der folgende Befehl macht das beispielhaft für den Benutzernamen `admin` und das Passwort `Geheimnis`:

```
htpasswd -nb admin Geheimnis | sed -e s/\\$/\\$\\$/g
```

Die Ausgabe dieses Befehls fügen Sie als Variable `http_basic_users` im Block `traefik` ein.

Sie können auch Ihren Passwortmanager benutzen, um sichere Passwörter für den Server zu erzeugen. Unter Windows und macOS würden wir von vornherein einen Passwortmanager wie KeePass [2] oder Bitwarden [3] zum Erzeugen der Passwörter empfehlen (beide laufen ebenfalls unter Linux). Da Sie die meisten Passwörter nie eingeben müssen, können Sie die extrem lang erzeugen lassen. Ausnahme: Das Passwort für BasicAuth brauchen Sie für jeden Login am Traefik-Dashboard.

Mit den Roles konfigurieren Sie, welche Dienste zu Ihrem Setup gehören. Dazu können Sie sich an unseren Vorlagen bedienen und Sub-Repositories in den Ordner roles/ klonen. Das geht mit `git submodule add` – ein vollständiges Beispiel hatten wir in [1] angegeben. Alternativ können Sie auch ohne Sub-Repository einfach Ordner in roles/ anlegen und selbst neue Dienste konfigurieren.

Die letzte Zutat sind Playbooks, die Sie im Wurzelverzeichnis Ihres Repository anlegen. Sie sind so kurz, dass es sich nicht lohnt, sie gesondert zu synchronisieren. Hier ein Beispiel für ein Playbook, das die Role für Traefik ausführt:

```
- hosts: server
  become: true
  roles:
    - role: traefik
      vars:
        service_cfg: "{{ traefik }}"
```

Die ersten drei Zeilen sind immer gleich. Danach kommt die YAML-Liste aller Roles, die das Playbook ausführt. Die Angabe hinter `vars:` setzt die Variablen aus dem `traefik`-Block definiert in group_vars/all.yml. Die Zeile weist der Variable `service_cfg` den Inhalt des Blocks zu und die Compose Hull ergänzt danach diesen Block. Wir nutzen solche nahezu identischen Playbooks mit genau einer Role, um Dienste bei Bedarf einzeln neu starten zu können, beispielsweise nachdem wir eine Variable geändert haben. Ein Playbook, das gleich mehrere Dienste aufsetzt, sähe genauso aus mit einer Liste mit mehr Spiegelstrichen hinter `roles:`.

„Aufsetzen“ und „Neustarten“ nutzen wir bei Ansible übrigens synonym. Ansible-Playbooks dienen nämlich immer dazu, einen bestimmten Zustand auf dem Server herzustellen. War ein Dienst dort nicht installiert, müssen die im Playbook definierten Roles alle Dateien und Ordner anlegen und den Dienst starten. Lief der Dienst bereits mit einer minimalen Änderung an der Konfiguration, überprüft Ansible nur, dass die Ordner und Dateien existieren und startet mit der geänderten Konfiguration neu. Wenn Sie selbst Roles schreiben, müssen Sie daran denken, dass die auch genauso funktionieren und nicht jeder zusätzliche Durchlauf einen bereits korrekten Zustand auf dem Server noch mal verändert.

Compose Hull

Der raffinierte Teil von Telerec't steckt in der Role `compose_hull`, die selbst nie in einem Playbook steht. Die Motivation: Wenn Sie im Netz auf die Suche nach attraktiven Serverdiensten gehen, werden Sie inzwischen fast immer auch Docker-Container und ein passendes docker-compose.yml finden. Wir wollten es so einfach wie möglich machen, Dienste mit diesen Infos auch mit Ansible zum Laufen zu bringen. Das Vorgehen dafür sieht folgendermaßen aus: Sie kopieren die Datei docker-compose.yml in eine leere Role (optional können Sie sie in den Unterordner templates/ stecken) und benennen sie um in docker-compose.yml.j2. Ab sofort ist sie ein Jinja2-Template, in dem Sie in doppelten geschweiften Klammern alle Ansible-Variablen benutzen können. So brauchen Sie kein Copy & Paste.

Gehen Sie dann die Datei durch und lagern Sie Geheimnisse wie Passwörter nach group_vars/all.yml in einen neuen Block mit dem Namen des Diensts aus. Kommen Werte in docker-compose.yml vor, die nicht geheim sind, erstellen Sie für diese Variablen in defaults/main.yml unterhalb von `service-defaults:` (siehe unten). So vermeiden Sie Fehler beim Copy & Paste, wenn sie die Werte mal anpassen müssen. Eine Variable `directory` referenzieren Sie im Template beispielsweise mit `{{ service_cfg.directory }}`.

Damit die Dienste vom Reverse-Proxy Traefik verschlüsselt und mit Zertifikaten ausgestattet werden, muss man eine ganze Reihe von Labels definieren und den Container, der von außen erreichbar ist, in das Docker-Netzwerk von Traefik einbuchen. Mit der Compose Hull geht das mit zwei simplen YAML-Referenzen auf vordefinierte Blöcke:

```
  labels: *base_labels
  networks: *base_networks
```

Falls Sie weitere Labels hinzufügen wollen, können Sie die YAML-Syntax mit `<<` verwenden:

```
  labels:
    << : *base_labels
        {# weitere.labels: "hier" #}
```

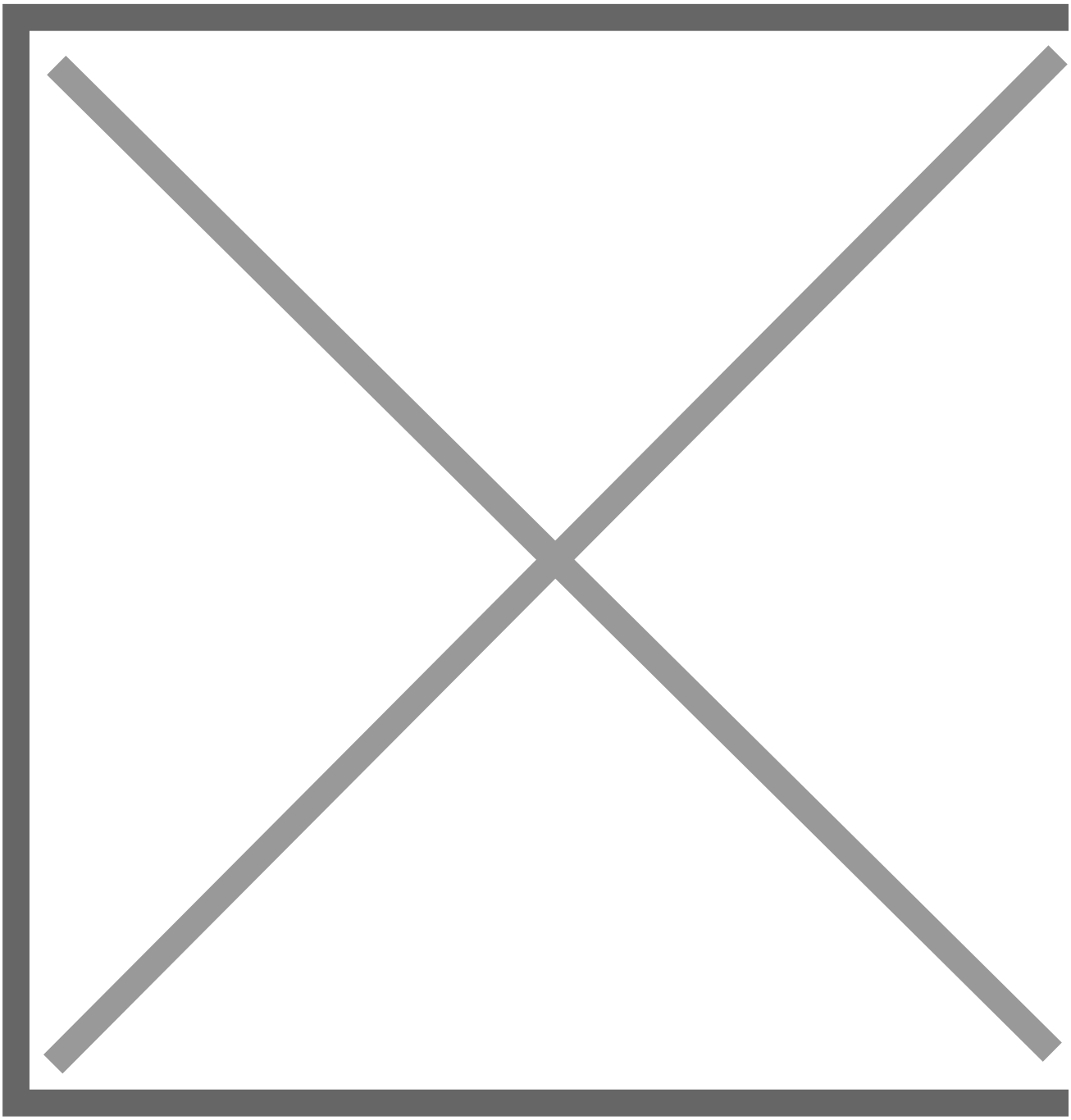
Um die Magie der Compose Hull zu nutzen, fügen Sie einen Task wie diesen in Ihre Role ein:

```
- ansible.builtin.import_role:
    name: compose_hull
  vars:
    service_defaults:
      directory: "{{ docker_dir }}/2fauth"
      name: twofauth
      traefik: true
      external: true
      watchtower: true
      autoheal: true
      skip_network_definition: false
```

Das Playbook dazu sieht dann wie folgt aus:

```
- hosts: server
  become: true
  roles:
    - role: twofauth
      vars:
        service_cfg: "{{twofauth_cfg}}"
```

Beim Aufruf fügt die Compose Hull zunächst alle Variablen in einen Block `service_cfg` zusammen. Entgegen Ansibles Voreinstellung überschreiben hier die Geheimnisse aus group_vars/all.yml die Voreinstellungen in der Role selbst. Einen Block mit passendem Namen (im Beispiel `twofauth_cfg`) muss es geben, er darf aber leer sein. Danach legt die Role falls nötig das `directory` und alle `subdirs` an. Dann baut die Compose Hull eine docker-compose.yml mit den Label- und Netzwerk-Blöcken und ersetzt dabei auch alle Jinja2-Variablen. Zuletzt führt der Task `community.docker.docker_compose` die Container-Konfiguration auf dem Server mit `docker-compose` hoch. Mit den Ansible-Tags `started`, `restarted`, `recreate` und `stopped` können Sie `docker-compose` auch Statusbefehle mitgeben. `started` ist die Standardeinstellung.



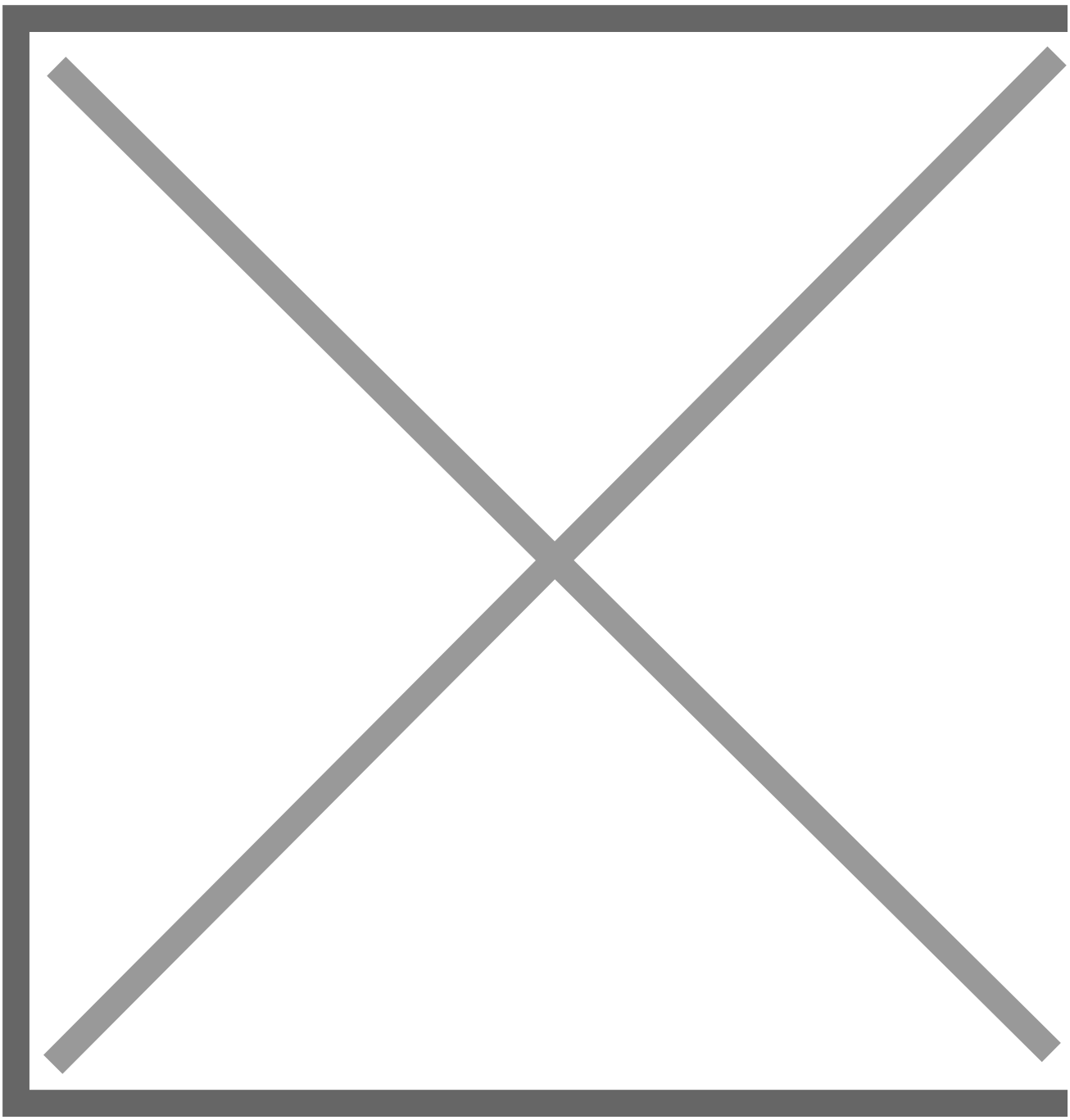
Traefik

Ein Proxy-Server vermittelt Anfragen zwischen Clients und Webservern. Ein Reverse Proxy macht das Gleiche auf der Serverseite. Er schaut bei Anfragen an die gleiche IP-Adresse nach, an welche Subdomain die eigentlich gerichtet waren und leitet die HTTP-Anfragen an den Webserver im passenden Container weiter.

In diesem Betriebsmodus läuft Traefik in Telerec’t. Zusätzlich kann er sich als zentraler Vermittler auch darum kümmern, Zertifikate auszuhandeln und den verschlüsselten Datenverkehr von den Clients schon mal zu entschlüsseln. Die Webserver in den Containern müssen dann keine

Zertifikate verwalten und bei unverschlüsselten Anfragen nicht auf HTTPS weiterleiten. Will man den Zugriff auf einen Webdienst mit HTTP-Basic-Authentifizierung einschränken, kann das Traefik mit einer Middleware namens BasicAuth übernehmen. Sie aktivieren diese, indem Sie einfach ein Label in der Container-Konfiguration ergänzen.

Um neue Dienste zu integrieren, müssen Sie die Konfigurationsdateien von Traefik nicht verändern. Stattdessen reagiert der Reverse Proxy auf bestimmte Labels, die direkt beim Container stehen. Das macht die Konfiguration angenehm modular. Mit einem Webinterface samt Dashboard informiert Traefik über die verwalteten Dienste und die jeweils genutzten Middlewares.

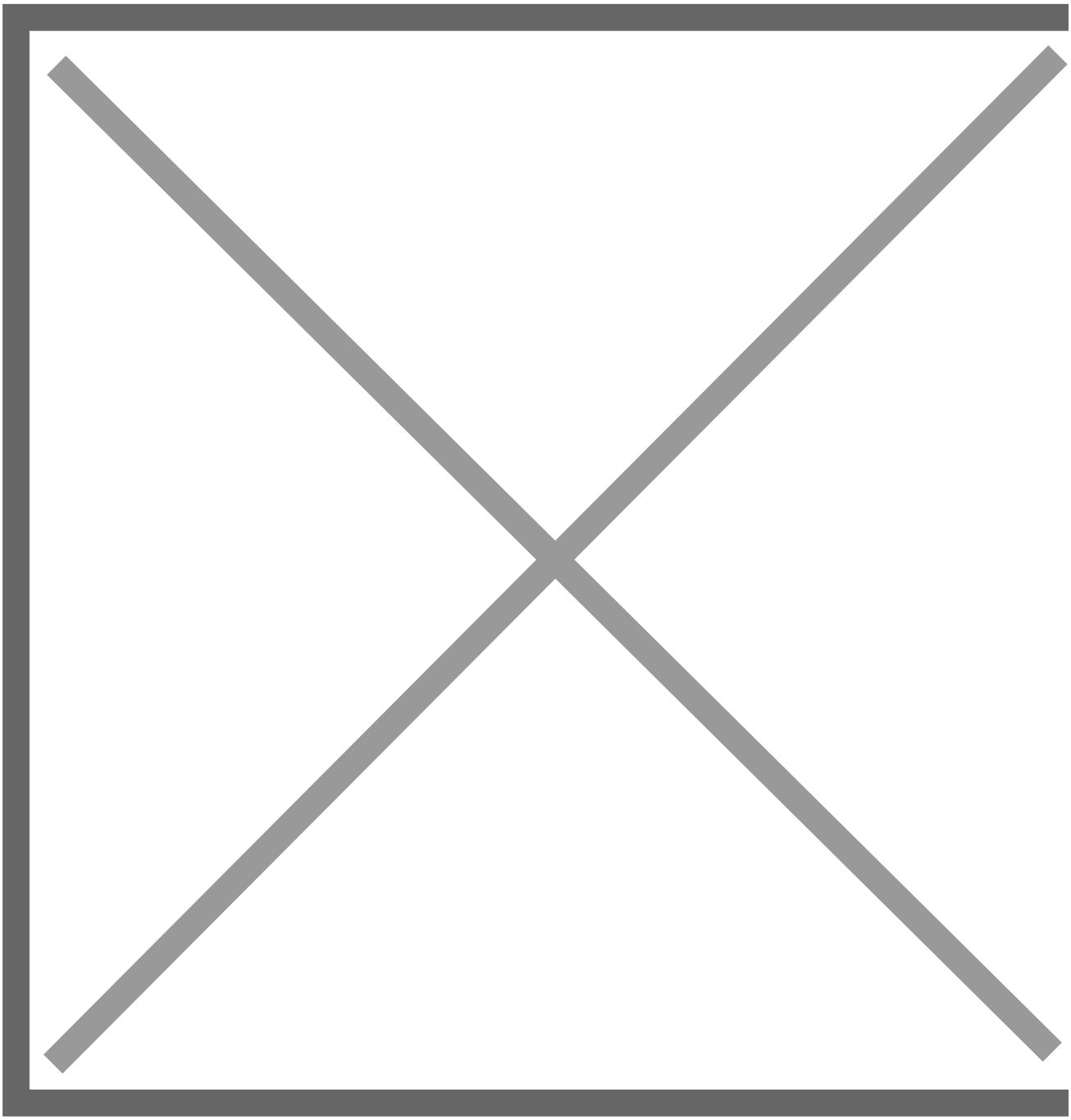


Watchtower und Autoheal

Auch für erfahrene Admins sind Sicherheitslücken höhere Gewalt. Weil es zu lange dauern würde, einen eigenen Patch zu entwickeln, warten Admins auf das nächste Update und spielen es sofort ein. Für große IT-Infrastruktur stehen dafür rund um die Uhr Admins bereit, beim eigenen Server ist die Gefahr aber groß, das Sicherheitsupdate erst verzögert einzuspielen.

Ein typisches Problem für eine weitere Schicht der Automatisierung: Watchtower überwacht die Registries, üblicherweise den Docker Hub, ob neue Images der installierten Container bereitstehen. Wenn ja, lädt die Software sie automatisch und tauscht sie im laufenden Betrieb aus. Dabei hält sich die Software an die im Compose-File definierten Tags, sodass man große Versionssprünge unterbinden kann, wenn man Probleme erwartet. Legen Sie die Tags aber nicht auf eine einzelne Version fest, weil das Sicherheitsupdates verhindern kann und setzen Sie auf Container, deren Maintainer Updates der Dienste auch verlässlich in die Images einbauen.

Autoheal kämpft ergänzend gegen Downtimes, indem es abgestürzte Container neu startet. Ob ein Container läuft, prüft ein hinter `healthcheck:` definierter Befehl. Das kann beispielsweise ein `wget`-Aufruf auf ein Webinterface sein. Gut programmierte automatische Tests können hier auch irreguläre Betriebszustände erkennen. Der Healthcheck prüft nur den Status-Code.



Portainer

Die Macht, neue Container ruckzuck auszuprobieren und mit einem einzigen Konsolenbefehl automatisiert aufzusetzen, wird Sie dazu verleiten, in kurzer Zeit viele Container anzusammeln. Konsolenbefehle wie `docker ps`, der alle laufenden Container auflistet, erfordern nicht nur ein SSH-Login, sie helfen auch kaum bei der Übersicht, wenn sich die Ausgabe über mehrere Bildschirmseiten erstreckt.

Mit Portainer kommt die Übersicht zurück. Die intuitive Weboberfläche informiert Sie, welche Container gerade laufen oder eben nicht. Praktisch ist, dass alle Container, die im gleichen Compose-File definiert wurden, auch in einen Stack gruppiert werden. Portainer zeigt mit wenigen Klicks Einstellungen und Logfiles und öffnet auf Wunsch direkt im Browserfenster eine Shell innerhalb eines Containers. Die Suche nach Problemen geht mit den von Portainer bereitgestellten Werkzeugen wesentlich flotter. Auch verwaiste Netzwerke und Images sind viel schneller aufgeräumt, als mit Dutzenden Konsolenbefehlen.

Portainer greift in Telerec't seine Infos über den Docker-Socket direkt ab, sodass Sie ohne weitere Konfiguration loslegen können. Ein Remote-Zugriff wäre mit der Software auch möglich, Telerec't braucht das aber nicht. Wer nicht alleine administriert, kann mit dem Admin-Account weitere Accounts anlegen. Das zwischen hell und dunkel einstellbare Theme passt sich automatisch der Systemeinstellung an.

Ausführung gefällig?

Unsere Erklärung der Basisdienste kommt ohne Installationsbefehle aus, weil wir alle vier Dienste bereits als Submodules im Basis-Repository definiert haben. Zur Erinnerung: Im ersten Teil des Artikels haben Sie die Submodules mit folgendem Befehl auf den Rechner geholt:

```
git submodule update --init \
  --recursive
```

Traefik-Konfiguration für Heimserver

Falls Sie für den Heimserver in den eigenen vier Wänden interne Dienste mit priviligiertem Zugriff aus dem Heimnetz einrichten möchten, geben Sie in der Liste `traefik.internal_ip_ranges` in group_vars/all.yml die IP-Bereiche an, in denen Ihr Router Adressen vergibt. Dienste mit der Variable `external: false` sind dann von Geräten mit internen IP-Adressen weiterhin erreichbar. Das bietet sich beispielsweise für Smart-Home-Container an, deren Ports Sie nicht im Internet veröffentlichen möchten. Alternativ können Sie diese Dienste auch extern freigeben und eine Authentifizierung wie bei der Weboberfläche von Traefik davor schalten.

Danach gibt es die Unterverzeichnisse in roles/ und Sie können die Roles in Playbooks benutzen. Die Playbooks für die vier Basisdienste haben wir im Basis-Repository schon vorbereitet und Sie können die vier Dienste nacheinander hochfahren:

```
pipenv run ansible-playbook traefik.yml -i hosts
pipenv run ansible-playbook watchtower.yml -i hosts
pipenv run ansible-playbook autoheal.yml -i hosts
pipenv run ansible-playbook portainer.yml -i hosts
```

All das geht alternativ auch in einem einzigen Befehl:

```
pipenv run ansible-playbook server-setup.yml -i hosts --ask-pass --ask-become-pass
```

Ihr Reich hat jetzt eine einwandfreie Infrastruktur. Sie dient allerdings bislang nur dem Zweck, sich selbst zu betreiben. Mit dieser Basis können Sie nun jedoch mit gutem Gewissen die Dienste installieren, für die Sie den Server eigentlich haben wollten. Nützlich finden wir beispielsweise: Nextcloud für Dateisynchronisation, Wordpress für einen privaten Blog, Vaultwarden für die Passwortverwaltung, NodeRed und Mosquitto für das Smart Home oder Wekan für private Kanban-Boards. Wenn wir in den folgenden Ausgaben solche Dienste mit Ansible aufsetzen, verweisen wir immer auf diesen Artikel, weil die Basis immer gleich ist, egal für welche dieser Dienste Sie sich entscheiden.

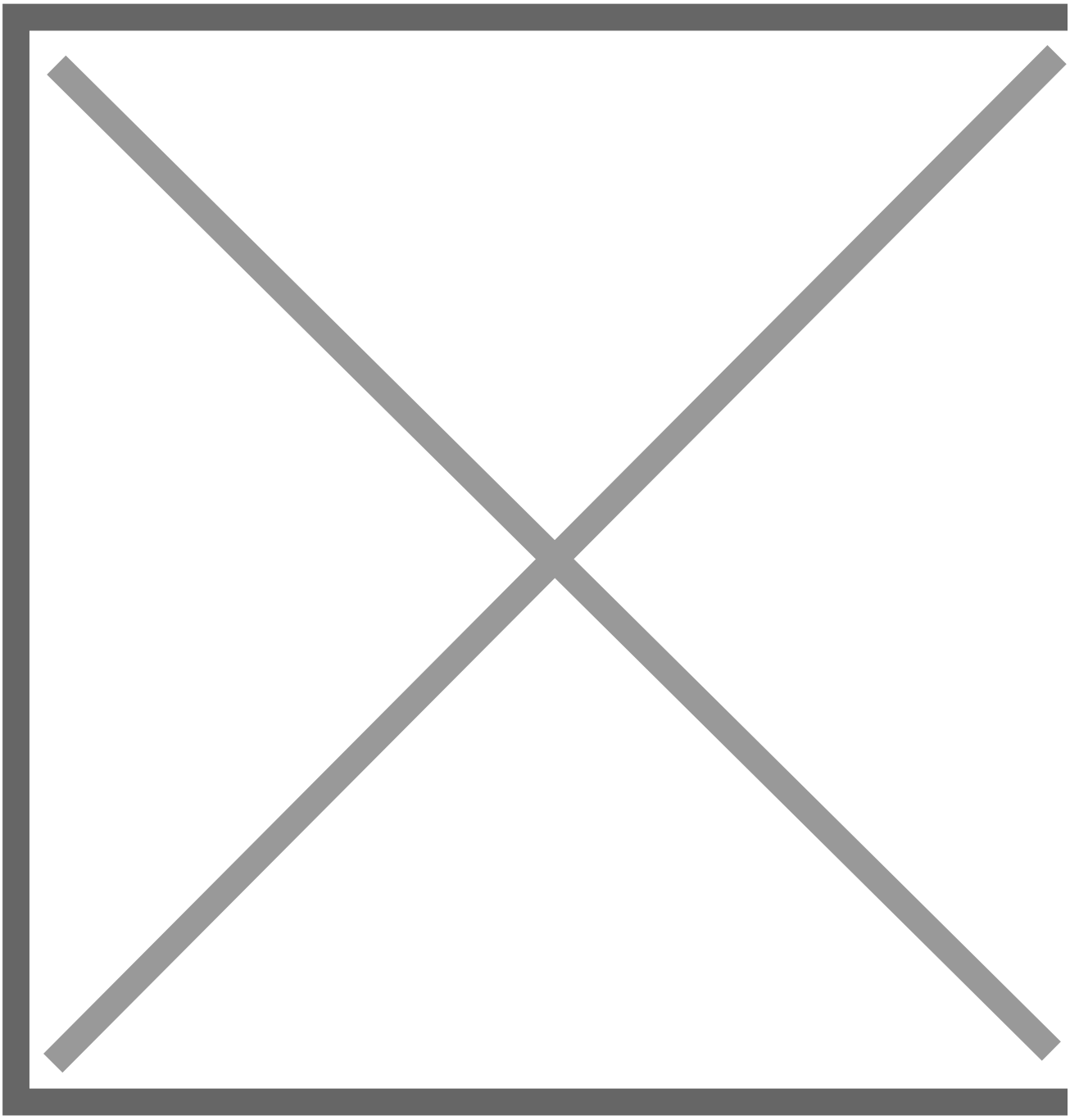
Unsere Hoffnung ist, dass unser Beispiel-Setup Telerec't Ihnen nicht nur Arbeit bei der Server-Administration abnimmt, sondern Sie auch Lust bekommen, das Setup mit eigenen Roles als Submodule zu erweitern. Von Ihren öffentlichen Repositories können dann auch andere profitieren und das Baukastensystem wächst und gedeiht. Wir verlinken Ihre Submodule-Repositories gern in der Readme-Datei unseres Basis-Repository. Schreiben Sie uns einfach eine kurze Nachricht mit dem Link an pmk@ct.de. *(pmk@ct.de)*

1. Literatur

2. Klaus Greff und Pina Merkert, Telerec't, Ein eigener Server im Rechenzentrum oder daheim – automatisiert mit Ansible, c't 1/2024, S. 150
3. Marvin Strathmann, Schlüsselfertig, So bringen Sie Ordnung ins Passwort-Chaos, c't 15/2019, S. 172
4. Niklas Dierking, Geheimniskrämer, Der Raspberry Pi als Passwort-Server, c't 9/2021, S. 18
5. Jan Mahn, Container-Komponist, Docker-Container mit Docker-Compose einrichten, c't 6/2018, S. 148
6. Jan Mahn, Container-Bedienpulte, Grafische Oberflächen für Docker, c't 6/2019, S. 158
7. Jan Mahn, HTTP-Einweiser, Eingehenden HTTP-Verkehr mit Traefik routen, c't 17/2019, S. 158
8. Merlin Schumacher, Container cum laude, Empfehlenswerte und gut gepflegte Docker-Container für den Alltag und als Inspiration, c't 16/2018, S. 108
9. Jan Mahn und Peter Siering, Container mit Docker und Docker-Compose, c't 14/2022, S. 186
10. Markus Stubbig, Cloudtresor zum Generieren von Einmalpasswörtern, 2FA-Authentifikator selbst gemacht, c't 27/2023, S. 146

Repositories: ct.de/yruq

Schaltzentrale für den Server



Weboberfläche für Ansible mit Semaphore

Pakete aktualisieren, Docker installieren, Server neu starten: Das praktische Kommandozeilenwerkzeug Ansible hilft Admins dabei, nervige Aufgaben zu automatisieren und versetzt Serverflotten in einen reproduzierbaren Zustand. Semaphore erweitert Ansible um eine praktische Weboberfläche.

Von Niklas Dierking
c't 3/2024 Seite 154

c't kompakt

- Ansible kümmert sich um Konfigurationsaufgaben und Automatisierung, hat aber eine steile Lernkurve.
- Semaphore stellt Ansible eine Weboberfläche zur Seite, die den Einstieg in Ansible erleichtert und dabei hilft, die eigene Serverflotte im Griff zu behalten.
- Wir zeigen, wie Sie Semaphore installieren, Tasks für Ansible erstellen und zu festgelegten Zeitpunkten ausführen.

Ansible kümmert sich zuverlässig um Konfigurationsaufgaben, indem es To-do-Listen abarbeitet, die im Ansible-Jargon Playbooks heißen. Auf einem frischen Server installieren Admins so in Windeseile wichtige Pakete und versetzen das System in einen klar definierten Zustand. In [1] und [2] lesen Sie eine Einführung in Ansible und wie Sie mit unserem Ansible-Projekt telerec’t eine Reihe von containerisierten Diensten auf einem Heim- oder Mietserver einrichten, die sich bewährt haben.

Üblicherweise läuft Ansible auf dem lokalen System des Administrators (Control-Host). Es kann es sich aber lohnen, Ansible auf ein dediziertes System auszulagern. Beispielsweise, wenn man möchte, dass der Server (Ziel-Host) jede Nacht prüft, ob es Updates gibt und diese einspielt. Als dedizierter Control-Host reicht eine schmale VM oder ein älterer Raspberry Pi in Ihrem Heimnetzwerk.

Das Open-Source-Projekt Semaphore erweitert Ansible um eine Weboberfläche, die als Schaltzentrale für Ihre Serverflotte dient. Für Nutzer, die mit der Kommandozeile weniger vertraut sind, erleichtert Semaphore den Einstieg in die Automatisierung mit Ansible. In diesem Artikel erfahren Sie, wie Sie Semaphore in Betrieb nehmen und damit Aufgaben auf entfernten Systemen ausführen. Ein Grundverständnis für Ansible und SSH ist dafür hilfreich.

Installation

Am einfachsten installieren Sie Semaphore auf einem Ubuntu-Host als Snap-Paket:

```
sudo snap install semaphore
```

Wir haben für unsere Testläufe Ubuntu Server 22.04 LTS genutzt, das wir von einem anderen Rechner im Netz via SSH bedienen. Snap hat den Vorteil, dass Ansible, die Datenbank BoltDB und weitere Abhängigkeiten mit im Snap-Container stecken und nicht zusätzlich installiert werden müssen. Wenn Sie Snap meiden oder eine andere Linux-Distribution als Ubuntu vorziehen, beschreibt die Semaphore-Dokumentation (ct.de/y4au) weitere Installationswege, beispielsweise mittels Docker, oder Sie laden ein Debian- oder RPM-Paket herunter, installieren Semaphore und eine kompatible Datenbank manuell.

Nach der Installation müssen Sie Semaphore stoppen und ein Benutzerkonto für den Administrator anlegen:

```
sudo snap stop semaphore

sudo semaphore user add --admin \
--login cttest \
--name=Testuser \
--email=cttest@example.com \
--password=geheim
```

Ersetzen Sie `cttest` durch einen eigenen Benutzernamen und `cttest@example.com` durch Ihre E-Mail-Adresse. Setzen Sie außerdem ein sicheres Passwort. Damit Ansible seine Arbeit verrichten kann, müssen Sie Zugangsdaten für die Ziel-Hosts in Semaphore hinterlegen. Eine ungeschützte Semaphore-Instanz dient Hackern als Generalschlüssel. Wir raten deswegen dazu, Semaphore nur im lokalen Netzwerk zu nutzen und nicht in das Internet zu hängen.

Starten Sie Semaphore mit `sudo snap start semaphore`, rufen in Ihrem Browser http://semaphore-host:3000 auf und melden sich dann mit den zuvor konfigurierten Zugangsdaten an. Ersetzen Sie semaphore-host durch den Hostnamen oder die IP-Adresse des Servers, der Semaphore ausführt.

Semaphore anfüttern

Als ersten Schritt legen Sie ein Projekt an und benennen es. Projekte dienen in Semaphore dazu, Automatisierungsaufgaben logisch zu trennen, beispielsweise eine Produktions- von einer Testumgebung. Sie können später beliebig viele weitere Projekte hinzufügen. Jetzt begrüßt Sie die Semaphore-Weboberfläche, die mit einigen Informationen gefüttert werden will, bevor Sie Ihr erstes Playbook ausführen.

Statten Sie zuerst dem Key Store einen Besuch ab, den Sie über die Seitenleiste auf der linken Seite des Fensters erreichen. Er verwaltet Zugangsdaten für Ziel-Hosts und Git-Repositories. Ansible führt Aufgaben auf Ziel-Hosts mittels SSH aus.

Dabei sollten Sie die Authentifizierung mittels SSH-Schlüsseln stets Passwörtern vorziehen. Damit das klappt, müssen Sie private SSH-Schlüssel in Semaphore hochladen, die als Gegenstück zu den öffentlichen Schlüsseln auf den Ziel-Hosts dienen. Schützen Sie Ihre Semaphore-Instanz gut. Eine kompromittierte Instanz ist eine Art Generalschlüssel für Angreifer.

Legen Sie einen Eintrag vom Typ „SSH Key“ an, fügen den privaten Schlüssel ein und geben ihm einen Namen. Wie Sie SSH-Schlüsselpaare erstellen und verwalten, lesen Sie in [3]. Die Anmeldung erfolgt standardmäßig mit dem Benutzer root. Wenn Ansible sich als ein anderer Benutzer anmelden soll, müssen Sie bei der Erstellung des Eintrags in Semaphore den korrekten Benutzer angeben.

Für Ziel-Hosts, auf die Sie via Passwort zugreifen, legen Sie einen Eintrag vom Typ „Login with password“ an und hinterlegen dann Nutzername und Passwort. Diese Zugangsdaten können auch als Passwort für sudo dienen, wenn Ansible sich als unprivilegierter Benutzer anmeldet und Aufgaben ausführen soll, die Systemverwalterrechte benötigen, beispielsweise bei der Installation von Paketen.

Legen Sie zuletzt noch einen Eintrag vom Typ „None“ an, dem Sie einen beliebigen Namen geben können. Semaphore verlangt, dass Sie jedem Git-Repository, aus dem es die Playbooks herunterlädt, ein Eintrag im Key Store zuordnen. Das gilt auch für öffentliche Repositories, wie unser Beispiel-Repository, das keine Zugangsdaten benötigt. Dafür brauchen Sie später den „None“-Schlüssel.

> Statt wie bei Ansible üblich auf der Kommandozeile, verwalten Sie Ihre Ziel-Hosts und die zuge
> Statt wie bei Ansible üblich auf der Kommandozeile, verwalten Sie Ihre Ziel-Hosts und die zugehörigen Zugangsdaten mit Semaphore über eine Weboberfläche.

Wechseln Sie anschließend zum Menü namens Inventory. Hier erstellen Sie eine Liste der Ziel-Hosts. Die Liste entspricht der Inventory-Datei, die Ansible gewöhnlich in /etc/ansible/hosts sucht oder deren Pfad Sie auf der Kommandozeile mit dem Parameter `-i` übergeben. Eine Inventory-Datei vom Typ „Static“ im Ini-Stil sieht beispielsweise so aus:

```
[hosts]
192.168.1.101
192.168.1.102
192.168.1.103
```

Zusätzlich müssen Sie das Inventory benennen und eine der zuvor konfigurierten Authentifizierungsmethoden angeben.

Wechseln Sie jetzt in das Environment-Menü. Environments enthalten Variablen, mit denen Sie Playbooks weiter anpassen können. Für jeden Task, den Sie mit Semaphore ausführen, müssen Sie ein Environment angeben, auch wenn Sie keine Variablen definieren wollen. Also richten Sie ähnlich wie beim „None“-Schlüssel ein leeres Environment ein. Tragen Sie dafür beim Erstellen einfach `{}` in den Feldern „Extra variables“ und „Environment variables“ ein und vergeben einen Namen.

Als letzte Zutat braucht Semaphore noch eine Quelle an Playbooks. Um die zu versionieren und gemeinsam zu bearbeiten ist es üblich, sie in GitHub-Repositories abzulegen. Damit Sie sich mit Semaphore vertraut machen können, haben wir ein öffentliches GitHub-Repository mit einer Reihe simpler Beispiel-Playbooks erstellt. Das können Sie natürlich auch forken, um die Playbooks anzupassen.

Legen Sie in Semaphore im Menü namens Repositories einen neuen Eintrag an. Vergeben Sie einen Namen, beispielsweise ansible-examples, fügen Sie die URL des Repository (https://github.com/ndi-ct/ansible-examples) oder Ihres Forks ein und weisen Semaphore an, den Branch namens „main“ zu nutzen. Weil es ein öffentliches Repository ist, reicht der zuvor konfigurierte Access Key „None“. Wenn Sie später mit Semaphore private Git-Repositories anzapfen wollen, müssen Sie den entsprechenden SSH-Schlüssel im Key Store hinterlegen.

To-do-Liste abhaken

Semaphore hat jetzt alle nötigen Informationen, damit Ansible loslegen kann. Erstellen Sie im Menü namens Task Templates ein neues Template, indem Sie auf „Create Template“ klicken. Für einen Testlauf bietet sich das integrierte Ping-Modul `ansible.builtin.ping` von Ansible an. Ping prüft, ob Ansible eine SSH-Verbindung zu den Zielservern aufbauen kann und ob Python installiert ist. Wenn nicht, gibt Ping eine Fehlermeldung aus.

Das Playbook mit dem Namen ping.yml umfasst nur wenige Zeilen YAML-Code und steckt mit im Beispiel-Repository:

```
---
- hosts: all
  tasks:
    - ansible.builtin.ping:
```

Semaphore unterteilt Task Templates mit drei verschiedenen Labels. „Task“ bietet sich für Administrations- und Konfigurationsaufgaben an. „Build“ ist für die Integration von Semaphore in CI/CD-Pipelines und „Deploy“ eignet sich für komplexere Softwareinstallationen, beispielsweise für einen Verbund von Docker-Containern, wie im telerec’t-Projekt.

Der Ping-Testlauf passt am besten zu „Task“. Geben Sie dem Task Template einen Namen, beispielsweise „Testlauf“. Außerdem müssen Sie den Namen des Playbooks (ping.yml), sowie das Inventory, das Repository und das Environment so wie im Screenshot auf Seite 155 definieren. Die restlichen Angaben sind optional, beispielsweise Kommandozeilenparameter (CLI Args) wie `--become`, wenn die Tasks Systemverwalterrechte benötigen.

> Task Templates stehen im Mittelpunkt von Semaphore und enthalten ein Playbook, ein Reposi
> Task Templates stehen im Mittelpunkt von Semaphore und enthalten ein Playbook, ein Repository, Variablen (Environment) und Ziel-Hosts (Inventory).

Klicken Sie in der Liste der Task Templates jetzt auf den Namen Ihres Templates und anschließend auf die Schaltfläche „Run“. Die zusätzlichen Optionen „Debug“, „Dry Run“ und „Diff“ können Sie erst mal ignorieren. Sie helfen bei Testläufen und der Fehlersuche, sollte ein Playbook mal nicht funktionieren.

Jetzt können Sie sich zurücklehnen und Ansible bei der Arbeit über die Schulter schauen. Zunächst fischt Ansible das Playbook ping.yml aus dem konfigurierten GitHub-Repository. Danach führt es den Task `ansible.builtin.ping` aus. Wurde der Task auf dem Zielserver erfolgreich abgeschlossen,

quittiert Ansible das in der Ausgabe mit `ok`:

```
TASK [ansible.builtin.ping] ***
ok: [192.168.1.101]
ok: [192.168.1.102]
ok: [192.168.1.103]
```

Wiederkehrende Aufgaben

Eine Semaphore-Instanz eignet sich besonders gut, um wiederkehrende Aufgaben auf Ziel-Hosts zu automatisieren. Um das zu zeigen, nutzen wir ein simples Playbook namens updates.yml, das mit dem Paketmanager apt prüft, ob neue Updates vorliegen, diese installiert und nicht mehr benötigte Pakete entfernt:

```
---
- name: Update packages via apt
  hosts: all
  gather_facts: true

  tasks:
    - name: Update package cache
      apt:
        update_cache: yes

    - name: Upgrade packages
      apt:
        upgrade: dist
        autoclean: yes
```

Erstellen Sie ein weiteres Task Template nach dem Vorbild des Ping-Beispiels, aber tragen Sie diesmal bei „Playbook Filename“ den Namen updates.yml ein. Um die Aufgabe zu terminieren, müssen Sie einen Zeitpunkt bei „Cron“ eintragen, beispielsweise `03***`, um die Aufgabe jeden Tag um 3 Uhr auszuführen. Wenn Sie Schwierigkeiten haben, Ihren Wunschzeitpunkt in eine Cron-Expression zu übersetzen, hilft das Onlinetool crontab guru, das wir unter ct.de/y4au verlinkt haben. Semaphore führt das Playbook ab jetzt stets zum konfigurierten Zeitpunkt aus.

> Wenn Sie einen Task in Semaphore anschieben, kann man Ansible in einem Ausgabefenster b
> Wenn Sie einen Task in Semaphore anschieben, kann man Ansible in einem Ausgabefenster bei der Arbeit zusehen.

Um Administratoren über den Status von Aufgaben zu informieren, kann Semaphore Nachrichten an Kanäle im Messenger Telegram verschicken. Das hat bei unserem Testlauf mit der Snap-Variante aber nicht funktioniert. Wenn Sie trotzdem auf dem Laufenden bleiben wollen, können Sie stattdessen eines der integrierten Benachrichtigungsmodule von Ansible nutzen (siehe ct.de/y4au).

Das Playbook mit dem Namen update-notification.yml in unserem Beispiel-Respository enthält eine Vorlage, um mittels Webhook eine Nachricht an einen Discord-Server zu schicken:

```
    - name: Discord-Notification
      community.general.discord:
        webhook_id: "id"
        webhook_token: "token"
        content: "Software update complete on {{ ansible_hostname }}."
```

Sie müssen lediglich die Platzhalter `id` und `token` durch die ID und das Token Ihres Discord-Webhook ersetzen. Die URL, die beide Werte enthält, zeigt Discord an, wenn Sie in den Kanaleinstellungen im Menü „Integration“ einen neuen Webhook erstellen. Die ID und das Token folgen auf `webhooks/` und werden durch `/`getrennt. Die Variable `{{ ansible_hostname }}` befüllt Ansible automatisch mit dem Hostnamen.

Fazit

Semaphore erweitert Ansible um eine grafische Benutzeroberfläche, mit der Sie sich im Handumdrehen eine Serverschaltzentrale einrichten. Das erleichtert den Einstieg in die Automatisierung mit Ansible und hilft dabei, die eigene Serverflotte zuverlässig in den gewünschten Zustand zu bringen. Wer eine Ansible-GUI möchte und wem Funktionen von Semaphore nicht mehr ausreichen, sollte einen Blick auf AWX werfen, das als Grundlage für die Red Hat Ansible Automation Platform dient, aber vorrangig für den Betrieb in einem Kubernetes-Cluster gedacht ist. *(ndi@ct.de*)

1. Literatur

2. Klaus Gref und Pina Merkert, Telerec’t, Server im Rechenzentrum oder daheim mit Ansible automatisieren, c’t 1/2024, S. 150
3. Klaus Gref und Pina Merkert, Ordnung im Königreich, Die Grundausstattung für öffentliche Server – automatisiert mit Ansible, c’t 02/2024, S. 154
4. Niklas Dierking, Schlüsselmeister, Sicher und komfortabel arbeiten mit SSH, c’t 21/2022, S. 172

GitHub-Repository ansible-examples, Semaphore-Dokumentation: ct.de/y4au

Git basics

Quick setup

Configure git

```
#
# Set user name and e-mail
# Change the default branch from master to main
# Store main repoitory hoster (like github) user credentials globally on next pull/push action
#
git config --global user.name "Mein Name"
git config --global user.email meine.mail@domain.de
git config --global init.defaultBranch main
git config --global credential.helper store
```

Create a new repository on the command line

```
#
# Assumes an empty Repository named my-repository on e.g. github.com
#
echo "# my-repository" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/rastef/my-repository.git
git push -u origin main
```

Push an existing repository from the command line

```
#
# Assumes an empty Repository named my-repository on e.g. github.com
#
git remote add origin https://github.com/rastef/my-repository.git
```

```
git branch -M main
git push -u origin main
```

Node.JS und Angular

Node JS

Node JS ist eine Voraussetzung für Angular. Welche Version von Node JS benötigt wird, kann der Angular Dokumentation entnommen werden. Es empfiehlt sich, immer die passende LTS-Version zu installieren. Um Node JS auf einem Linux-Rechner zu installieren, sollte zuerst der Node Version Manager ***nvm*** installiert werden. Das kann unter Ubuntu durch Herunterladen und Ausführen eines Shell-Skriptes erledigt werden:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.1/install.sh | bash
```

Anschließend sollt bei Verbleib im Terminal einmal `source .bashrc` ausgeführt werden. Mit nvm kann nun die gewünschte Node JS Version installiert werden. Für die aktuelle LTS-Version 20 lautet der Befehl `nvm install 20`. MIt `npm -v` kann die installierte Node Version kontrolliert werden.

1blu KVM-Installation

Installation von Ubuntu auf einem vServer via KVM

Vorbereitung

Im ersten Schritt legt man im Menu Neuinstallation unter Betriebssystem (leere KVM-Instanz) eine neue leere Instanz für das Betriebssystem Ubuntu an.

Datenbanken

Datenbanken

PostgreSQL mit homebrew

Installation:

```
brew update && brew upgrade
brew install prostgresql@<version>
```

Erstkontakt:

```
psql -h localhost -U postgres
```

Wenn hierbei der Fehler auftaucht, das die Rolle *postgres* unbekannt ist, muss zuerst der Superuser prostgres angelegt werden:

```
createuser -s postgres
psql -h localhost -U postgres
```

Der Befehl *createuser* gehört zum PostgreSQL-Umfang. Sollte es Probleme beim Aufruf geben, muss der PATH angepasst werden:

```
echo 'export PATH="/usr/local/opt/postgresql@15/bin:$PATH"' >> ~/.zshrc
```