

Kubernetes

- [Installation](#)
 - [Kubernetes Cluster installieren \(Baremetal\)](#)
 - [Installation von MetallLB \(Lokaler LoadBalancer\)](#)
 - [K3s Cluster installieren](#)
 - [Traefik-Reverseproxy vom K3s-Cluster entfernen](#)
 - [Metrics-Server vom K3s-Cluster entfernen](#)
- [Befehle](#)
 - [Cheat Sheet: Namespaces in Kubernetes](#)
 - [Cheat Sheet: Pods in Kubernetes](#)
- [Manifest-Dateien](#)
 - [Grundgerüst Kubernetes Namespace YAML-Datei](#)
 - [Manifest Dateien anwenden oder löschen](#)
 - [Grundgerüst Kubernetes Deployment YAML-Datei](#)
 - [Grundgerüst Kubernetes Service YAML-Datei](#)
 - [Service Externer-Zugriff Typ](#)
 - [Grundgerüst Kubernetes Ingress YAML-Datei](#)
- [Storage / Daten Speicherung](#)
 - [Kubernetes Daten auf Samba Freigabe speichern](#)
 - [NFS-Freigabe als PVC-Speicher im Kubernetes Pod verwenden](#)
- [Deployments](#)

Installation

Kubernetes Cluster installieren (Baremetal)

Einleitung

In diesem Artikel geht es darum, wie wir einen Kubernetes Cluster aufsetzen können, um von den Funktionalitäten von Kubernetes zu profitieren. Kubernetes ist mittlerweile eine weitverbreitete Orchestrationsplattform für Container.

Voraussetzungen

Um ein Kubernetes Cluster zu installieren und dieser Anleitung zu folgen, müssen die folgenden Voraussetzungen erfüllt sein:

- Mindestens Server mit einem installierten **Debian 11** oder **Debian 12**
- Pro Server mindestens **2 virtuelle CPU-Kerne**
- Pro Server mindestens **2 GB Arbeitsspeicher**
- Pro Server mindestens **20 GB freier Festplattenplatz**
- Administrationsbenutzer
- Stabile Internetverbindung

Installation

Netzwerkdesign

In dieser Anleitung werden wir 2 Server betreiben. Einer wird als Master Node fungieren, und der zweite Server als Worker Node. Der Master Node ist für die Verwaltung des Kubernetes Clusters zuständig. Der Worker Node führt nur die sogenannten Pods auf.

Hostbeschreibung	Hostname	IP-Adresse
Master Node	srv-kub-master	192.168.10.200
Worker Node	srv-kub-worker1	192.168.10.201

Installation

Im ersten Schritt installieren wir ein paar benötigte Pakete und deaktivieren auf jedem Node den Swap-Speicher. Dies ist zwar nicht zwingend erforderlich, aber funktioniert in der Regel besser.

```
apt install -y sudo curl socat -y
sudo swapoff -a
```

```
sudo sed -i ' / swap / s/^\(.*\)$/#\1/g' /etc/fstab
```

Im nächsten Schritt installieren wir die containerd Laufzeitumgebung und stellen ein paar Dinge ein. Dazu führen wir die folgenden Befehle aus. Zuerst stellen wir ein paar Kernel Parameter ein:

```
cat <<EOF | sudo tee /etc/modules-load.d/containerd.conf
overlay
br_netfilter
EOF
sudo modprobe overlay
sudo modprobe br_netfilter
cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-k8s.conf
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF
```

Um die Änderungen jetzt zu übernehmen für wir den folgenden Befehl aus:

```
sudo sysctl --system
```

Jetzt installieren wir das Paket containerd und stellen wieder ein paar Dinge ein.

```
sudo apt update
sudo apt -y install containerd
containerd config default | sudo tee /etc/containerd/config.toml >/dev/null 2>&1
```

Im nächsten Schritt setzen wir den "*cgroupdriver*" auf allen Nodes.

```
sudo nano /etc/containerd/config.toml
```

Dort müssen wir in dem Pfad

`[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]` die Option `SystemdCgroup` auf `true` verändern.

Danach starten wir den Dienst von containerd einmal neu.

```
sudo systemctl restart containerd
sudo systemctl enable containerd
```

Im nächsten Schritt fügen wir das Kubernetes Apt Repository hinzu.

```
echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg]
https://pkgs.k8s.io/core:/stable:/v1.28/deb/ /" | sudo tee
/etc/apt/sources.list.d/kubernetes.list
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.28/deb/Release.key | sudo gpg --dearmor -o
/etc/apt/keyrings/kubernetes-apt-keyring.gpg
```

Jetzt installieren wir die Kubernetes Tools auf unseren Servern.

```
sudo apt update
sudo apt install kubelet kubeadm kubectl -y
sudo apt-mark hold kubelet kubeadm kubectl
```

Zum Testen, ob alles geklappt hat, können wir einmal `kubectl version` ausführen.

Kubernetes Konfiguration

Jetzt konfigurieren wir unseren Kubernetes Cluster und verbinden die einzelnen Hosts miteinander, um die grundlegenden Funktionen von Kubernetes zu erhalten.

Wir erstellen im ersten Schritt eine yaml Datei, welches die Konfiguration von unserem Cluster enthält und passen diese Datei unserem Belieben an.

Dieser Schritt muss nur auf dem Master Node durchgeführt werden!

```
nano kubelet.yaml
```

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: InitConfiguration
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
kubernetesVersion: "1.28.0" # Ersetzen mit deiner eingesetzten Version
controlPlaneEndpoint: "k8s-master"
---
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
```

Nachdem wir die Datei erstellt haben, initialisieren wir jetzt unser Kubernetes Cluster.

```
sudo kubeadm init --config kubelet.yaml
```

Wenn alles geklappt hat, sollte eine Meldung auftauchen, dass das **Control-Plane** erfolgreich initialisiert wurde. Wenn dies der Fall ist, sehen wir auch die entsprechenden Befehle damit die anderen **Worker Nodes** als **Worker** oder als **Master** beitreten können. Diese Befehle können wir bei Bedarf auch neu erstellen. Für das erste die Befehle zur Seite kopieren.

Wir müssen im Anschluss noch einmal die Befehle ausführen, die benötigt werden, damit mit dem Control-Plane kommuniziert werden kann.

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Um zu testen, ob der Cluster richtig hochgefahren wurde, können wir die folgenden Befehle ausführen.

```
kubectl get nodes
kubectl cluster-info
```

Jetzt können wir auf den Worker Nodes die Befehle ausführen, um dem Kubernetes Cluster beizutreten. Wenn die Nodes beigetreten sind, sollten diese mit dem Befehl `kubectl get nodes` ersichtlich sein. Damit die Nodes im Status hochgefahren werden, brauchen wir sogenannte Netzwerk Add-ons. Wir verwenden hier Calico.

Um Calico zu installieren, führen wir den folgenden Befehl aus:

```
kubectl apply -f
https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/calico.yaml
```

Um zu überprüfen, ob die Calico Pods laufen, führen wir den folgenden Befehl aus:

```
kubectl get pods -n kube-system
```

Kubernetes Cluster testen

Um das Kubernetes Cluster zu testen, führen wir den folgenden Befehl auf dem Master Node aus:

```
kubectl create deployment nginx-app --image=nginx --replicas 2
kubectl expose deployment nginx-app --name=nginx-web-svc --type NodePort --port 80
kubectl describe svc nginx-web-svc
```

Jetzt sollten wir, wenn alles geklappt hat, eine Ausgabe mit dem entsprechenden Port erhalten. Das heißt, wenn wir eine Webanfrage auf die externe IP mit dem angegebenen Port starten, sollte uns die "NGINX Welcome Page" begrüßen.

Mit dem folgenden Befehl können wir das gestartete Deployment wieder stoppen und löschen:

```
kubectl delete deployment nginx-app
```

Installation von MetalLB (Lokaler LoadBalancer)

Einleitung

In dieser kurzen Anleitung beschreibe ich kurz, wie wir mit der Hilfe von MetalLB einen lokalen LoadBalancer betreiben können. In der Regel verwendet man einen LoadBalancer innerhalb eines Cloud-Providers und dieser stellt dann einen kostenpflichtigen LoadBalancer zur Verfügung. Sobald wir aber im LAN einen solchen LoadBalancer über die Service-Konfiguration anfordern, sollte die Anfrage auf `"pending"` stehen bleiben und keine IP erhalten. Mit diesem LoadBalancer sind die internen Anwendungen dann erreichbar über eine dedizierte IP-Adresse.

Durchführung

Im ersten Schritt führen wir den folgenden Befehl auf unserem Kubernetes Master Node aus:

```
kubectl apply -f  
https://raw.githubusercontent.com/metallb/metallb/v0.14.8/config/manifests/metallb-native.yaml
```

Damit wird dann die Konfiguration für den LoadBalancer Pod heruntergeladen und gestartet. Um zu überprüfen, ob die Pods erfolgreich gestartet wurden, können wir den folgenden Befehl ausführen:

```
kubectl get pods --namespace metallb-system
```

Wenn hier bei allen Containern der Status auf `Running` steht, sollten die Pods ordnungsgemäß hochgefahren sein. Wir müssen dann im Anschluss eine neue YAML-Datei anlegen, in dem wir den IPv4-Bereich definieren, welcher vom LoadBalancer verwendet werden darf. Die Datei sieht folgendermaßen aus:

```
apiVersion: metallb.io/v1beta1  
kind: IPAddressPool  
metadata:  
  name: <pool-name>  
  namespace: metallb-system  
spec:  
  addresses:  
    - <ip-von_ip-bis>
```


Jetzt müssen wir eine weitere YAML-Datei erstellen. Diese enthält das "L2Advertisement" und enthält den entsprechenden IP-Pool, der zur Vergabe der IP-Adressen verwendet werden darf. Die Datei sieht wie folgt aus:

```
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: l2metallb
  namespace: metallb-system
spec:
  ipAddressPools:
    - <pool-name>
```

Diese beiden Konfigurationen werden dann auch wieder einmal über `kubectl apply -f <dateiname>` aktiviert. Wenn alles geklappt hat, können wir in der Service-Konfiguration den Typen auf LoadBalancer setzen. Wenn wir jetzt die Service-Konfiguration aktualisieren, sollten wir mit dem folgenden Befehl dann die IP-Adresse unseres LoadBalancers sehen können.

```
kubectl get services --all-namespaces
```

K3s Cluster installieren

Einleitung

In dieser kurzen Anleitung beschreibe ich, wie wir einen K3s-Cluster installieren können. Auf diesem Wege können wir viel schneller ein laufendes Kubernetes-Cluster auf die Beine stellen. K3s wird als fertiges Skript bereitgestellt, welches auf Ressourcensparenden Betrieb ausgelegt ist.

Installation

Um das Cluster zu installieren, benötigen wir im ersten Schritt 3 eigenständige Linux-Server. In meinem Fall sind das alle Debian 12 Server mit jeweils einer eigenen IP-Adresse.

Um den Master-Node zu installieren, führen wir den folgenden Befehl aus:

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

Sobald das Skript durchgelaufen ist, läuft unser Control-Plane-Server unseres Kubernetes Cluster. Dieses können wir überprüfen, indem wir den folgenden Befehl ausführen, um alle momentan verfügbaren Nodes unseres Clusters einzusehen:

```
kubectl get nodes
```

Jetzt wollen wir im nächsten Schritt unsere Worker-Nodes dem Cluster hinzufügen. Dazu benötigen wir im ersten Schritt den Cluster-Token. Diesen finden unter: `/var/lib/rancher/k3s/server/node-token`. Der Inhalt der Datei sieht wie folgt aus:

```
K10b02c9d094b28ce8099ca6bbded97e68f7734af130f8a19103cdc7dfc8bf89cda::server:  
4f03364535d090536b282a9d2d22681a
```

Es wird hier aber nur das gelb hinterlegte (Ab `server:`) benötigt. Das ist der Cluster-Token. Jetzt verbinden wir uns auf einen Worker-Node und führen den folgenden Befehl aus:

```
curl -sfL https://get.k3s.io | K3S_URL=https://<ip-master-node>:6443 K3S_TOKEN=<cluster-token>  
sh -
```

Info: Es muss hier noch die IP-Adresse oder DNS-Name des Master-Nodes und der Cluster-Token eingetragen werden.

Sobald das Skript durchgelaufen ist, sollte der Node dem Cluster beigetreten sein. Dies können wir überprüfen, indem wir auf dem Master wieder den `kubectl get nodes` ausführen. Wenn hier der

Worker Node auftaucht, hat alles wie gewünscht geklappt. Diese Schritte führen wir dann für weitere Worker Nodes aus. So können wir unser Cluster nach Belieben erweitern.

Traefik-Reverseproxy vom K3s-Cluster entfernen

Einleitung

In diesem Artikel geht es kurz darum, wie wir in unserem K3s Cluster den Traefik-Reverseproxy entfernen können, welcher standardgemäß immer mitinstalliert wird.

Durchführung

Um Traefik zu entfernen, müssen wir die service-Datei des K3s-Dienstes anpassen. Dazu verbinden wir uns auf unseren Master-Node und öffnen die folgende Datei:

```
nano /etc/systemd/system/k3s.service
```

Dort fügen wir unter `ExecStart` noch `--disable=traefik` ein. Das sollte dann wie folgt aussehen:

```
ExecStart=/usr/local/bin/k3s \  
    server \  
    --disable=traefik \  
    --
```

Im Anschluss erstellen wir noch eine andere Datei mit dem folgenden Befehl:

```
touch /var/lib/rancher/k3s/server/manifests/traefik.yaml.skip
```

Zum Schluss starten wir jetzt einmal alle Server neu. Dann sollte der Traefik-Pod nicht mehr gestartet werden.

Metrics-Server vom K3s-Cluster entfernen

Einleitung

In diesem Artikel geht es kurz darum, wie wir in unserem K3s Cluster den Metrics-Server entfernen können, welcher standardgemäß immer mitinstalliert wird.

Durchführung

Um den Metrics-Server zu entfernen, müssen wir die service-Datei des K3s-Dienstes anpassen. Dazu verbinden wir uns auf unseren Master-Node und öffnen die folgende Datei:

```
nano /etc/systemd/system/k3s.service
```

Dort fügen wir unter `ExecStart` noch `--disable=traefik` ein. Das sollte dann wie folgt aussehen:

```
ExecStart=/usr/local/bin/k3s \  
server \  
--disable=metrics-server \  

```

Im Anschluss erstellen wir noch eine andere Datei mit dem folgenden Befehl:

```
touch /var/lib/rancher/k3s/server/manifests/traefik.yaml.skip
```

Zum Schluss starten wir jetzt einmal alle Server neu. Dann sollte der Traefik-Pod nicht mehr gestartet werden.

Befehle

Cheat Sheet: Namespaces in Kubernetes

Alle Namespaces anzeigen

Um alle verfügbaren Namespaces unseres Kubernetes Clusters anzuzeigen, führen wir den folgenden Befehl aus:

```
kubectl get namespaces
```

Neuen Namespace anlegen

Um einen neuen Namespace im Kubernetes Cluster anzulegen, führen wir den folgenden Befehl aus:

```
kubectl create namespace <name>
```

Namespace löschen

Um einen Namespace im Kubernetes Cluster zu löschen, führen wir den folgenden Befehl aus:

```
kubectl delete namespace <name>
```

Informationen über Namespace einsehen

Wenn du Informationen über deinen Namespace erhalten möchtest, verwende den folgenden Befehl:

```
kubectl describe namespace <name>
```

Cheat Sheet: Pods in Kubernetes

Alle Pods in einem Namespace anzeigen

Mit dem folgenden Befehl werden alle Pods angezeigt, die in einem bestimmten Namespace einen Status haben:

```
kubectl get pods --namespace <namespace name>
```

Alle Pods im Cluster anzeigen

Mit dem folgenden Befehl werden alle Pods angezeigt, die auf unserem Kubernetes Cluster laufen:

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

Ausführliche Informationen über einen Pod anzeigen

Mit dem folgenden Befehl werden weitere Informationen über einen Pod angezeigt und können eingesehen werden. Hier sehen wir diverse Tags oder z.B. auf welchem Node der Pod momentan läuft.

```
kubectl describe pod <pod-name> --namespaces <namespace>
```

Konsole eines Pods aufrufen

Mit dem folgenden Befehl können wir eine interaktive Konsole in unserem Pod öffnen, um z.B. besseres Debugging durchzuführen:

```
kubectl exec -it <pod-name> --namespace <namespace name> -- sh
```


Manifest-Dateien

Grundgerüst Kubernetes Namespace YAML-Datei

```
apiVersion: v1
kind: Namespace
meta:
  name: einnamespace
  labels:
    # Maschinen auswertbar
    author: phillip
    name: appl
    <key>: <value>
  annotations:
    # Menschen auswertbar
    author: Phillip <mail@phillipunzen.de>
```

Manifest Dateien anwenden oder löschen

Um Manifest-Dateien in Kubernetes anzuwenden oder zu aktualisieren, verwenden wir den folgenden Befehl:

```
kubectl apply -f <Pfad zur Datei>
```

Um Manifest-Dateien aus dem Kubernetes Cluster zu entfernen, verwenden wir den folgenden Befehl:

```
kubectl delete -f <Pfad zur Datei>
```

Grundgerüst Kubernetes Deployment YAML-Datei

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: <deployment name>
  namespace: <namespace angeben>
  annotations:
    author: Phillip <mail@phillipunzen.de>

spec:
  replicas: 3
  strategy:
    type: <Typ> RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  selector:
    matchLabels:
      app: <app-name>
  template:
    metadata:
      labels:
        app: <app-name>
      annotations:
        author: Phillip <mail@phillipunzen.de>
    spec:
      containers:
        - name: <pod name>
          image: <docker-image:tag>
          env:
            - name: KEY
              value: VALUE
          ports:
            - name: http
```

```
    containerPort: <Port im Container>
resources:
  requests: # Minimum an Hardware
    cpu: "250m" # => 1/4 CPU Kern für den Container
    memory: "256Mi"
  limits:
    cpu: "1000m" # => 1 CPU Kern für den Container
    memory: "512Mi" # => Wenn mehr benötigt, dann wird Container neugestartet
readinessProbe: # Healthcheck auf HTTP
  httpGet:
    path: /
    port: http
  initialDelaySeconds: 10 # Zeitraum zwischen Checks
livenessProbe:
  httpGet:
    path: /
    port: http
  initialDelaySeconds: 10 # Zeitraum zwischen Checks
```

Grundgerüst Kubernetes Service YAML-Datei

```
apiVersion: v1
kind: Service
metadata:
  name: <app-name>
  namespace: <namespace name>
  ...
spec:
  selector:
    #Selektor definieren
    app: <app-name>
  ports:
    - name: http
      port: 80
      targetPort: http (targetPort Namen aus dem Deployment)
  type: <Typ>
```

Service Externer-Zugriff Typ

ClusterIp	Innerhalb des Clusters
NodePort	Port wird auf einem Port der Nodes geöffnet
LoadBalancer	Verteilt die Anfragen auf die Pods (Liegt außerhalb des Kubernetes Clusters) => Kostet in der Cloud extra

Grundgerüst Kubernetes Ingress YAML-Datei

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: <Name>
  namespace: <Namespace>
spec:
  ingressClassName: <Name> nginx
  rules:
    - host: <dns name> website.domain.de
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: <app name>
                port:
                  name: <Port Name> http
```


Storage / Daten Speicherung

Kubernetes Daten auf Samba Freigabe speichern

Einleitung

In dieser Anleitung geht es darum, wie wir Daten aus unserem Kubernetes Cluster auf einer Samba-Freigabe speichern können. So können Daten, die von Containern generiert werden, z.B. auf einem zentralen Fileserver gespeichert werden. In der Regel verwendet man für Kubernetes eine NFS-Freigabe, da Kubernetes in der Regel auf Linux-Servern läuft. Die Einrichtung erfolgt grob in 6 Schritten:

1. Namespace erstellen
2. RBAC Ressourcen erstellen (Berechtigungen)
3. CSI-SMB-Treiber installieren
4. CSI-SMB-Controller ausrollen
5. CSI-SMB Node Daemon installieren
6. SMB-Secret erstellen

Durchführung

Namespace erstellen

Um einen Namespace zu erstellen, können wir entweder direkt über `kubectl` einen Namespace erstellen, oder wir definieren den Namespace über eine YAML-Datei.

Wenn wir direkt über `kubectl` einen Namespace erstellen möchten, verwenden wir den folgenden Befehl:

```
kubectl create namespace smb-provisioner
```

Falls wir doch den Weg über die Yaml-Datei gehen möchten, verwenden wir die folgende Yaml-Datei und aktivieren im Anschluss die Datei über den gewohnten Weg über den `kubectl apply` Befehl.

```
apiVersion: v1
kind: Namespace
metadata:
  name: smb-provisioner
```

Im Anschluss können wir mit dem folgenden Befehl überprüfen, ob der Namespace angelegt wurde.

```
kubectl get namespaces
```

RBAC Ressourcen erstellen

In diesen Schritt erstellen wir die benötigten RBAC-Ressourcen. Diese dienen dazu, die Berechtigungen mit unserem Kubernetes-Cluster zu vereinen. Mit diesen können dann die Container auf die entsprechenden Samba-Freigaben zugreifen und dort Daten ablegen oder lesen.

Wir erstellen jetzt eine Yaml-Datei welches die `ServiceAccounts`, eine `ClusterRole` und eine `ClusterRoleBinding` enthält. Der Inhalt der Yaml-Datei sieht wie folgt aus:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: csi-smb-controller-sa
  namespace: smb-provisioner
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: csi-smb-node-sa
  namespace: smb-provisioner
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: smb-external-provisioner-role
rules:
  - apiGroups: [""]
    resources: ["persistentvolumes"]
    verbs: ["get", "list", "watch", "create", "delete"]
  - apiGroups: [""]
    resources: ["persistentvolumeclaims"]
    verbs: ["get", "list", "watch", "update"]
  - apiGroups: ["storage.k8s.io"]
    resources: ["storageclasses"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["events"]
    verbs: ["get", "list", "watch", "create", "update", "patch"]
  - apiGroups: ["storage.k8s.io"]
    resources: ["csinodes"]
```

```

  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["coordination.k8s.io"]
  resources: ["leases"]
  verbs: ["get", "list", "watch", "create", "update", "patch"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: smb-csi-provisioner-binding
subjects:
- kind: ServiceAccount
  name: csi-smb-controller-sa
  namespace: smb-provisioner
roleRef:
  kind: ClusterRole
  name: smb-external-provisioner-role
  apiGroup: rbac.authorization.k8s.io

```

Wir aktivieren im Anschluss diese Datei wieder mit `kubectl apply`. Damit sollten dann die entsprechenden **ServiceAccounts** und **Cluster-Rollen** erstellt werden, die benötigt werden.

CSI-SMB-Treiber Installation

In diesem Schritt installieren wir jetzt den CSI-SMB-Treiber. Dieser wird zur Interaktion zwischen dem Kubernetes-Cluster und dem Samba-Protokoll benötigt. Dazu legen wir wieder eine Yaml-Datei mit dem folgenden Inhalt an und aktivieren diese danach wieder.

```

apiVersion: storage.k8s.io/v1
kind: CSIDriver
metadata:
  name: smb.csi.k8s.io
spec:
  attachRequired: false
  podInfoOnMount: true

```

Wir können hier auch wieder einmal überprüfen, ob alles geklappt hat, mit dem folgenden Befehl:

```
kubectl get csidrivrs.storage.k8s.io
```

CSI-SMB Controller ausrollen

In diesem Schritt richten wir den CSI-SMB-Controller ein, welcher auch benötigt wird. Dazu erstellen wir wieder eine Yaml-Datei und aktivieren diese nach dem Erstellen:

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: csi-smb-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      app: csi-smb-controller
  template:
    metadata:
      labels:
        app: csi-smb-controller
    spec:
      dnsPolicy: Default # available values: Default, ClusterFirstWithHostNet, ClusterFirst
      serviceAccountName: csi-smb-controller-sa
      nodeSelector:
        kubernetes.io/os: linux
      priorityClassName: system-cluster-critical
      tolerations:
        - key: "node-role.kubernetes.io/master"
          operator: "Exists"
          effect: "NoSchedule"
        - key: "node-role.kubernetes.io/controlplane"
          operator: "Exists"
          effect: "NoSchedule"
        - key: "node-role.kubernetes.io/control-plane"
          operator: "Exists"
          effect: "NoSchedule"
      containers:
        - name: csi-provisioner
          image: registry.k8s.io/sig-storage/csi-provisioner:v3.2.0
          args:
            - "-v=2"
```

```

- "--csi-address=$(ADDRESS)"
- "--leader-election"
- "--leader-election-namespace=kube-system"
- "--extra-create-metadata=true"
env:
  - name: ADDRESS
    value: /csi/csi.sock
volumeMounts:
  - mountPath: /csi
    name: socket-dir
resources:
  limits:
    cpu: 1
    memory: 300Mi
  requests:
    cpu: 10m
    memory: 20Mi
- name: liveness-probe
  image: registry.k8s.io/sig-storage/livenessprobe:v2.7.0
  args:
    - --csi-address=/csi/csi.sock
    - --probe-timeout=3s
    - --health-port=29642
    - --v=2
  volumeMounts:
    - name: socket-dir
      mountPath: /csi
  resources:
    limits:
      cpu: 1
      memory: 100Mi
    requests:
      cpu: 10m
      memory: 20Mi
- name: smb
  image: registry.k8s.io/sig-storage/smbplugin:v1.9.0
  imagePullPolicy: IfNotPresent
  args:
    - "--v=5"
    - "--endpoint=$(CSI_ENDPOINT)"
    - "--metrics-address=0.0.0.0:29644"

```

```
ports:
  - containerPort: 29642
    name: healthz
    protocol: TCP
  - containerPort: 29644
    name: metrics
    protocol: TCP
livenessProbe:
  failureThreshold: 5
  httpGet:
    path: /healthz
    port: healthz
  initialDelaySeconds: 30
  timeoutSeconds: 10
  periodSeconds: 30
env:
  - name: CSI_ENDPOINT
    value: unix:///csi/csi.sock
securityContext:
  privileged: true
volumeMounts:
  - mountPath: /csi
    name: socket-dir
resources:
  limits:
    memory: 200Mi
  requests:
    cpu: 10m
    memory: 20Mi
volumes:
  - name: socket-dir
    emptyDir: {}
```

Um zu überprüfen, ob hier auch wieder alles läuft, führen wir den folgenden Befehl aus:

```
kubectl -n csi-smb-provisioner get deploy,po,rs -o wide
```

CSI-SMB-Node Daemon installieren

Um jetzt den CSI-SMB-Node Daemon zu installieren, erstellen wir wieder eine Yaml-Datei mit dem folgenden Inhalt und aktivieren diese wieder.

```
kind: DaemonSet
apiVersion: apps/v1
metadata:
  name: csi-smb-node
spec:
  updateStrategy:
    rollingUpdate:
      maxUnavailable: 1
    type: RollingUpdate
  selector:
    matchLabels:
      app: csi-smb-node
  template:
    metadata:
      labels:
        app: csi-smb-node
    spec:
      hostNetwork: true
      dnsPolicy: Default # available values: Default, ClusterFirstWithHostNet, ClusterFirst
      serviceAccountName: csi-smb-node-sa
      nodeSelector:
        kubernetes.io/os: linux
      priorityClassName: system-node-critical
      tolerations:
        - operator: "Exists"
      containers:
        - name: liveness-probe
          volumeMounts:
            - mountPath: /csi
              name: socket-dir
          image: registry.k8s.io/sig-storage/livenessprobe:v2.7.0
          args:
            - --csi-address=/csi/csi.sock
            - --probe-timeout=3s
            - --health-port=29643
            - --v=2
          resources:
            limits:
              memory: 100Mi
            requests:
```



```

    cpu: 10m
    memory: 20Mi
- name: node-driver-registrar
  image: registry.k8s.io/sig-storage/csi-node-driver-registrar:v2.5.1
  args:
    - --csi-address=$(ADDRESS)
    - --kubelet-registration-path=$(DRIVER_REG_SOCK_PATH)
    - --v=2
  livenessProbe:
    exec:
      command:
        - /csi-node-driver-registrar
        - --kubelet-registration-path=$(DRIVER_REG_SOCK_PATH)
        - --mode=kubelet-registration-probe
    initialDelaySeconds: 30
    timeoutSeconds: 15
  env:
    - name: ADDRESS
      value: /csi/csi.sock
    - name: DRIVER_REG_SOCK_PATH
      value: /var/lib/kubelet/plugins/smb.csi.k8s.io/csi.sock
  volumeMounts:
    - name: socket-dir
      mountPath: /csi
    - name: registration-dir
      mountPath: /registration
  resources:
    limits:
      memory: 100Mi
    requests:
      cpu: 10m
      memory: 20Mi
- name: smb
  image: registry.k8s.io/sig-storage/smbplugin:v1.9.0
  imagePullPolicy: IfNotPresent
  args:
    - "--v=5"
    - "--endpoint=$(CSI_ENDPOINT)"
    - "--nodeid=$(KUBE_NODE_NAME)"
    - "--metrics-address=0.0.0.0:29645"
  ports:

```

```
- containerPort: 29643
  name: healthz
  protocol: TCP
livenessProbe:
  failureThreshold: 5
  httpGet:
    path: /healthz
    port: healthz
  initialDelaySeconds: 30
  timeoutSeconds: 10
  periodSeconds: 30
env:
- name: CSI_ENDPOINT
  value: unix:///csi/csi.sock
- name: KUBE_NODE_NAME
  valueFrom:
    fieldRef:
      apiVersion: v1
      fieldPath: spec.nodeName
securityContext:
  privileged: true
volumeMounts:
- mountPath: /csi
  name: socket-dir
- mountPath: /var/lib/kubelet/
  mountPropagation: Bidirectional
  name: mountpoint-dir
resources:
  limits:
    memory: 200Mi
  requests:
    cpu: 10m
    memory: 20Mi
volumes:
- hostPath:
    path: /var/lib/kubelet/plugins/smb.csi.k8s.io
    type: DirectoryOrCreate
  name: socket-dir
- hostPath:
    path: /var/lib/kubelet/
    type: DirectoryOrCreate
```

```
    name: mountpoint-dir
  - hostPath:
      path: /var/lib/kubelet/plugins_registry/
      type: DirectoryOrCreate
    name: registration-dir
```

SMB-Secret erstellen

In diesem Schritt erstellen wir jetzt einen SMB-Secret. Dieser wird zur Authentifizierung am Samba-Server benötigt. Es werden jetzt die Anmeldeinformationen eines Benutzers für die Samba-Freigabe benötigt.

Im ersten Schritt erstellen wir hier einen Namespace für unsere Anwendung.

```
kubectl create namespace test
```

Jetzt erstellen wir einen Secret mit den Anmeldeinformationen unseres erstellten Benutzers. Hier müssen die Platzhalter mit den entsprechenden Informationen noch ausgetauscht werden.

```
kubectl -n test create secret generic smb-creds \
--from-literal username=<username> \
--from-literal domain=<domain> \
--from-literal password=<password>
```

Damit wird ein Secret erstellt, welchen wir dann innerhalb unseres Kubernetes-Clusters abrufen können.

PersistentVolume erstellen

In diesem Schritt erstellen wir jetzt das PersistentVolume. Dieses kann man als Speicherplatzreservierung für das gesamte Kubernetes-Cluster verstehen. Damit teilen wir Kubernetes mit, wo das Cluster Daten ablegen kann. Dafür erstellen wir jetzt wieder eine Yaml-Datei und aktivieren diese wieder mit `kubectl apply`:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-smb
  namespace: test
spec:
  storageClassName: ""
  capacity:
    storage: <größe> #50Gi
  accessModes:
```

```

- ReadWriteMany
persistentVolumeReclaimPolicy: Retain
mountOptions:
- dir_mode=0777
- file_mode=0777
- vers=3.0
csi:
  driver: smb.csi.k8s.io
  readOnly: false
  volumeHandle: <volume-name> # Eindeutige Bezeichnung im Cluster
  volumeAttributes:
    source: <server-freigabepfad> #Pfad der Samba Freigabe mit rekursiven Ordnern
  nodeStageSecretRef:
    name: smb-creds
    namespace: test

```

Auch hier können wir wieder die Durchführung mit dem folgenden Befehl überprüfen:

```
kubectl -n test get pv
```

Jetzt erstellen wir das **PersistentVolumeClaim** welches die Speicherplatzreservierung für eine einzelne Anwendung darstellt. Hier kommunizieren wir mit dem Kubernetes-Cluster und sagen diesem, wie viel Speicherplatz unsere Anwendung im Kubernetes-Cluster benötigt. Dazu erstellen wir wieder eine neue Yaml-Datei mit dem folgenden Inhalt und aktivieren diese im Anschluss wieder:

```

kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-smb
  namespace: test
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: <größe> #10Gi
  volumeName: pv-smb
  storageClassName: ""

```

Um jetzt zu überprüfen, ob die Schreiberechtigungen vorliegen, kann das nachstehende Deployment verwendet werden. Dieses erstellt eine einfache Datei, die den aktuellen Timestamp

hereinschreibt. So können wir testen, dass alles klappt, wie wir uns das vorstellen.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: deploy-smb-pod
  namespace: test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    name: deploy-smb-pod
    spec:
      nodeSelector:
        "kubernetes.io/os": linux
      containers:
        - name: deploy-smb-pod
          image: mcr.microsoft.com/oss/nginx/nginx:1.19.5
          command:
            - "/bin/bash"
            - "-c"
            - set -euo pipefail; while true; do echo $(date) >> /mnt/smb/outfile; sleep 1;
done
      volumeMounts:
        - name: smb
          mountPath: "/mnt/smb"
          readOnly: false
      volumes:
        - name: smb
          persistentVolumeClaim:
            claimName: pvc-smb
```

Wenn jetzt eine entsprechende Datei erstellt wird, scheint alles zu klappen und wir können unsere Daten auf einer Samba-Freigabe ablegen.

NFS-Freigabe als PVC-Speicher im Kubernetes Pod verwenden

Einleitung

In diesem kurzen Artikel geht es darum, wie wir eine **NFS-Freigabe** als **persistenten (PVC) Speicher** verwenden können. Dies verwenden wir z.B. wenn wir keinen Cloud Kubernetes Cluster betreiben, sondern diesen Cluster bei uns im lokalen LAN betreiben.

Über den NFS-Share stellen wir dann sicher, dass jeder Pod Zugriff auf dieselben Daten hat. Ansonsten müssten wir auf einem anderen Weg sicherstellen, dass auf jedem Node die Daten vorhanden sind.

Info: Stelle bitte sicher das das Paket `nfs-common` bereits auf allen Nodes installiert ist!

Durchführung

Helm Installation

Im ersten Schritt müssen wir auf unserem **Master Node** einmal den **Helm Paketmanager** installieren. Dazu führen wir den folgenden Befehl aus:

```
curl -L https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | bash
```

Um zu überprüfen, ob die Installation geklappt hat, kannst du den folgenden Befehl ausführen:

```
helm version
```

Installation vom nfs-provisioner

Im nächsten Schritt installieren wir den **nfs-subdir-external-provisioner**. Dies ist eine Speicherklasse, um den **NFS-Speicher** in das **Kubernetes Cluster** einzubinden. Kubernetes hat von Haus aus keine Möglichkeit, **NFS-Speicher** anzubinden. Um die Installation durchzuführen und einen dedizierten Namespaces zu erstellen, führe die folgenden Befehle aus:

```
helm repo add nfs-subdir-external-provisioner https://kubernetes-sigs.github.io/nfs-subdir-external-provisioner/  
kubectl create namespace nfs-provisioner
```

Starten des nfs-provisioner Pods

In diesem Schritt starten wir den Pod für den **NFS-Provisioner**. Dazu passen wir noch die Serveradresse und den Pfad unserer NFS-Freigabe an.

```
helm install nfs-client nfs-subdir-external-provisioner/nfs-subdir-external-provisioner --set  
nfs.server=<server-adresse> --set nfs.path=<freigabe-pfad> --namespace <namespace-name>
```

Anlegen der Manifest-Dateien für den Storage

Im nächsten Schritt erstellen wir eine **YAML-Datei** für das "*PersistentVolume*". Diese enthält den folgenden Inhalt:

```
apiVersion: v1  
kind: PersistentVolume  
metadata:  
  name: nfs-pv  
  namespace: nfs-storage  
spec:  
  capacity:  
    storage: <größe> (z.B. 450Gi)  
  accessModes:  
    - ReadWriteMany  
  nfs:  
    server: <server-adresse>  
    path: <freigabe-pfad>
```

Jetzt legen wir die **YAML-Datei** für das "*PersistentVolumeClaim*" an. Dies erhält die entsprechende Größe, welches unsere Anwendung benötigt. Die Datei sieht wie folgendermaßen aus:

```
apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
  name: nfs-pvc  
  namespace: nfs-storage  
spec:  
  accessModes:  
    - ReadWriteMany  
  resources:  
    requests:  
      storage: <größe>
```

Info zu PV und PVC: Ein PV wird vom Administrator definiert und präsentiert das Speichermedium (Speicher Backend) und das PVC stellt die Anfrage vom Pod an den PV um

die entsprechende Größe für die App vom Volume zu erhalten. In der Regel erstellt man einen PV pro Speichermedium und einen PVC pro Applikation / Pod.

Im nächsten Schritt müssen wir beide Dateien wie gewohnt mit `kubectl apply -f <dateipfad>` aktivieren.

Pod Konfiguration anpassen

Um jetzt den Speicher in dem Pod verfügbar zu machen, müssen wir in der **Deployment-Datei** unseres Pods im spec Segment die **volumes** definieren.

```
volumes:
  - name: nfs-volum-app
    nfs:
      server: <server-adresse>
      path: <freigabe-pfad>
```

Dann können wir jetzt im Container Segment in **volumeMounts** definieren:

```
volumeMounts:
  - mountPath: <container pfad>
    name: <name-des-volumes>
```

Wenn wir jetzt im Anschluss den Pod starten, und die Berechtigungen stimmen, dann sollten die Daten geschrieben werden und damit sind jetzt die Daten persistent verfügbar.

Beispiel Pod-Konfiguration

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nextcloud
  namespace: nextcloud
  annotations:
    author: Phillip
  labels:
    app: nextcloud
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nextcloud
```

```
template:
  metadata:
    labels:
      app: nextcloud
  spec:
    containers:
      - name: nextcloud-app
        image: lscr.io/linuxserver/nextcloud:latest
        env:
          - name: PUID
            value: "1000"
          - name: GUID
            value: "1000"
        ports:
          - name: http
            containerPort: 80
        resources:
          requests:
            cpu: "250m"
            memory: "256Mi"
          limits:
            cpu: "2000m"
            memory: "4096Mi"
        volumeMounts:
          - mountPath: /config
            name: nfs-volume-nextcloud-config
          - mountPath: /data
            name: nfs-volume-nextcloud-data
    volumes:
      - name: nfs-volume-nextcloud-config
        nfs:
          server: 192.168.5.4
          path: /mnt/Kubernetes/Daten/nextcloud-config
      - name: nfs-volume-nextcloud-data
        nfs:
          server: 192.168.5.4
          path: /mnt/Kubernetes/Daten/nextcloud-data
```

apiVersion: v1

kind: Service

```
metadata:
  name: nextcloud-service
  namespace: nextcloud
spec:
  selector:
    app: nextcloud
  ports:
    - port: 80

---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nextcloud
  namespace: nextcloud
spec:
  ingressClassName: nginx
  rules:
    - host: nextcloud.k8s.domain.de
      http:
        paths:
          - pathType: Prefix
            path: "/"
            backend:
              service:
                name: nextcloud-service
                port:
                  number: 80
```

Deployments