

Storage / Daten Speicherung

- Kubernetes Daten auf Samba Freigabe speichern
- NFS-Freigabe als PVC-Speicher im Kubernetes Pod verwenden

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.

Quelle: github.io

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> --namespaces <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
```

