Using Rook / Ceph with PVCs on Azure Kubernetes Service

#ceph #kubernetes #rook #aks

Christian Dennig

Dec 28, 2019 *Originally published at partlycloudy.blog on Dec 8, 2019* · 10 min read

Introduction

As you all know by now, Kubernetes is a quite popular platform for running cloud-native applications at scale. A common recommendation when doing so, is to ousource as much state as possible, because managing state in Kubernetes is not a trivial task. It can be quite hard, especially when you have a lot of attach/detach operations on your workloads. Things can go terribly wrong and – of course – your application and your users will suffer from that. A solution that becomes more and more popular in that space is Rook in combination with Ceph.

Rook is described on their homepage rook.io as follows:

Christian Dennig

Dev. OSS. Kubernetes. Cloud Native. CKAD. Cloud Solution Architect @Microsoft. I've done everything wrong first so that you don't have to. BTW: #fcknzs

Follow

WORK
Cloud Solution Architect at Microsoft

LOCATION
Germany

JOINED
Dec 27, 2019

More from Christian Dennig

Getting started with KrakenD on Kubernetes / AKS
#apigateway #azure #kubernetes #microservices

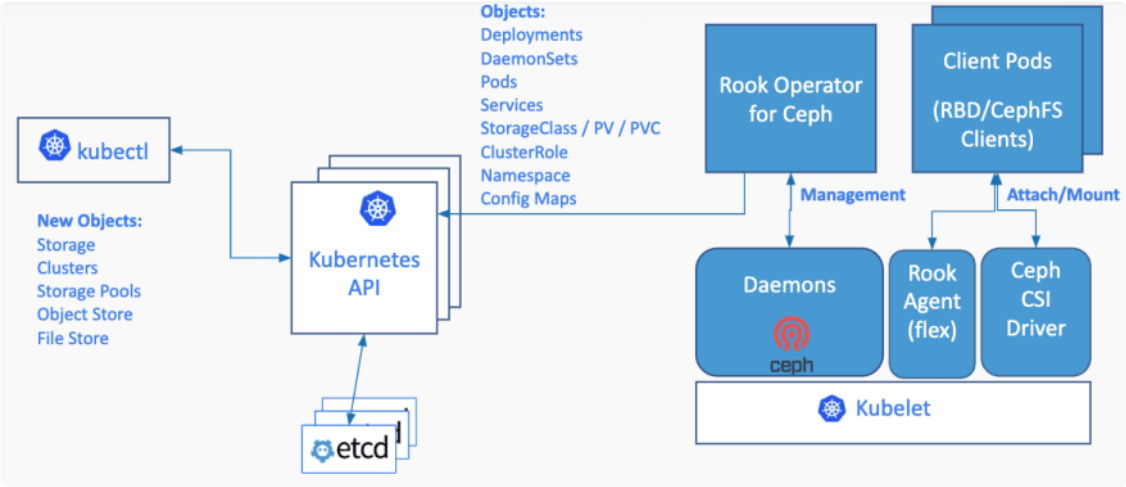
Release to Kubernetes like a Pro with Flagger
#deployment #flagger #kubernetes #canary

> Rook turns distributed storage systems into self-managing, self-scaling, self-healing storage services. It automates the tasks of a storage administrator: deployment, bootstrapping, configuration, provisioning, scaling, upgrading, migration, disaster recovery, monitoring, and resource management.

Rook is a project of the Cloud Native Computing Foundation, at the time of writing in status “incubating”.

Ceph in turn is a free-software storage platform that implements storage on a cluster, and provides interfaces for object-, block- and file-level storage. It has been around for many years in the open-source space and is a battle-proven distributed storage system. Huge storage systems have been implemented with Ceph.

So in a nutshell, Rook enables Ceph storage systems to run on Kubernetes using Kubernetes primitives. The basic architecture for that inside a Kubernetes cluster looks as follows:



Rook in-cluster architecture

Horizontal Autoscaling in Kubernetes #3 – KEDA

#keda #kubernetes #autoscaling #prometheus

I won't go into all of the details of Rook / Ceph, because I'd like to focus on simply running and using it on AKS in combination with PVCs. If you want to have a step-by-step introduction, there is a pretty good "Getting Started" video by Tim Serewicz on Vimeo:

First, we need a Cluster!

So, let's start by creating a Kubernetes cluster on Azure. We will be using different nodepools for running our storage (nodepool: *npstorage*) and application workloads (nodepool: *npstandard*).

```
# Create a resource group

$ az group create --name rooktest-rg --location westeurope
```

```
# Create the cluster

$ az aks create \
--resource-group rooktest-rg \
--name myrooktestclstr \
--node-count 3 \
--kubernetes-version 1.14.8 \
--enable-vmss \
--nodepool-name npstandard \
--generate-ssh-keys
```

Add Storage Nodes

After the cluster has been created, add the *npstorage* nodepool:

```
$ az aks nodepool add --cluster-name myrooktestclstr \
--name npstorage --resource-group rooktest-rg \
--node-count 3 \
--node-taints storage-node=true:NoSchedule
```

Please be aware that we add **taints** to our nodes to make sure that no pods will be scheduled on this nodepool as long as we explicitly tolerate it. We want to have these nodes exclusively for storage pods!

> If you need a refresh regarding the concept of “taints and tolerations”, please see the Kubernetes documentation.

So, now that we have a cluster and a dedicated nodepool for storage, we can download the cluster config.

```
```

```
$ az aks get-credentials \
--resource-group rooktest-rg \
--name myrooktestclstr
```

Let's look at the nodes of our cluster:

```
$ kubectl get nodes

NAME                                STATUS   ROLES   AGE    VE
aks-npstandard-33852324-vmss000000  Ready    agent   10m    v1
aks-npstandard-33852324-vmss000001  Ready    agent   10m    v1
aks-npstandard-33852324-vmss000002  Ready    agent   10m    v1
aks-npstorage-33852324-vmss000000   Ready    agent   2m3s   v1
aks-npstorage-33852324-vmss000001   Ready    agent   2m9s   v1
aks-npstorage-33852324-vmss000002   Ready    agent   119s   v1
```

So, we now have three nodes for storage and three nodes for our application workloads. From an infrastructure level, we are now ready to install Rook.

Install Rook

Let's start installing Rook by cloning the repository from GitHub:

```
$ git clone https://github.com/rook/rook.git
```

After we have downloaded the repo to our local machine, there are three steps we need to perform to install Rook:

1. Add Rook CRDs / namespace / common resources

2. Add and configure the Rook operator
3. Add the Rook cluster

So, switch to the */cluster/examples/kubernetes/ceph* directory and follow the steps below.

1. Add Common Resources

```
$ kubectl apply -f common.yaml
```

The *common.yaml* contains the namespace *rook-ceph,* common resources (e.g. clusterroles, bindings, service accounts etc.) and some Custom Resource Definitions from Rook.

2. Add the Rook Operator

The operator is responsible for managing Rook resources and needs to be configured to run on Azure Kubernetes Service. To manage Flex Volumes, AKS uses a directory that's different from the "default directory". So, we need to tell the operator which directory to use on the cluster nodes.

Furthermore, we need to adjust the settings for the CSI plugin to run the corresponding daemonsets on the storage nodes (remember, we added taints to the nodes. By default, the pods of the daemonsets Rook needs to work, won't be scheduled on our storage nodes – we need to "tolerate" this).

So, here's the full operator.yaml file

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rook-ceph-operator
  namespace: rook-ceph
  labels:
    operator: rook
    storage-backend: ceph
spec:
  selector:
    matchLabels:
      app: rook-ceph-operator
  replicas: 1
  template:
    metadata:
      labels:
        app: rook-ceph-operator
    spec:
      serviceAccountName: rook-ceph-system
      containers:
      - name: rook-ceph-operator
        image: rook/ceph:master
        args: ["ceph", "operator"]
        volumeMounts:
        - mountPath: /var/lib/rook
          name: rook-config
        - mountPath: /etc/ceph
          name: default-config-dir
        env:
        - name: ROOK_CURRENT_NAMESPACE_ONLY
          value: "false"
        - name: FLEXVOLUME_DIR_PATH
```

```
    value: "/etc/kubernetes/volumeplugins"

  - name: ROOK_ALLOW_MULTIPLE_FILESYSTEMS
    value: "false"
  - name: ROOK_LOG_LEVEL
    value: "INFO"
  - name: ROOK_CEPH_STATUS_CHECK_INTERVAL
    value: "60s"
  - name: ROOK_MON_HEALTHCHECK_INTERVAL
    value: "45s"
  - name: ROOK_MON_OUT_TIMEOUT
    value: "600s"
  - name: ROOK_DISCOVER_DEVICES_INTERVAL
    value: "60m"
  - name: ROOK_HOSTPATH_REQUIRES_PRIVILEGED
    value: "false"
  - name: ROOK_ENABLE_SELINUX_RELABELING
    value: "true"
  - name: ROOK_ENABLE_FSGROUP
    value: "true"
  - name: ROOK_DISABLE_DEVICE_HOTPLUG
    value: "false"
  - name: ROOK_ENABLE_FLEX_DRIVER
    value: "false"
  # Whether to start the discovery daemon to watch for ra
  # This daemon does not need to run if you are only goin
  - name: ROOK_ENABLE_DISCOVERY_DAEMON
    value: "false"
  - name: ROOK_CSI_ENABLE_CEPHFS
    value: "true"
  - name: ROOK_CSI_ENABLE_RBD
    value: "true"
  - name: ROOK_CSI_ENABLE_GRPC_METRICS
    value: "true"
  - name: CSI_ENABLE_SNAPSHOTTER
```

```
    value: "true"

  - name: CSI_PROVISIONER_TOLERATIONS
    value: |
      - effect: NoSchedule
        key: storage-node
        operator: Exists
  - name: CSI_PLUGIN_TOLERATIONS
    value: |
      - effect: NoSchedule
        key: storage-node
        operator: Exists
  - name: NODE_NAME
    valueFrom:
      fieldRef:
        fieldPath: spec.nodeName
  - name: POD_NAME
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
  - name: POD_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
volumes:
- name: rook-config
  emptyDir: {}
- name: default-config-dir
  emptyDir: {}
```

3. Create the Cluster

Deploying the Rook cluster is as easy as installing the Rook operator. As we are running our cluster with the Azure Kuberntes Service – a

As we are running our cluster with the Azure Kubernetes Service – a managed service – we don't want to manually add disks to our storage nodes. Also, we don't want to use a directory on the OS disk (which most of the examples out there will show you) as this will be deleted when the node will be upgraded to a new Kubernetes version.

In this sample, we want to leverage Persistent Volumes / Persistent Volume Claims that will be used to request Azure Managed Disks which will in turn be dynamically attached to our storage nodes. Thankfully, when we installed our cluster, a corresponding storage class for using Premium SSDs from Azure was also created.

```
$ kubectl get storageclass

NAME                PROVISIONER                AGE
default (default)   kubernetes.io/azure-disk   15m
managed-premium     kubernetes.io/azure-disk   15m
```

Now, let's create the Rook Cluster. Again, we need to adjust the tolerations and add a node affinity that our OSDs will be scheduled on the storage nodes:

```
apiVersion: ceph.rook.io/v1
kind: CephCluster
metadata:
  name: rook-ceph
  namespace: rook-ceph
spec:
  dataDirHostPath: /var/lib/rook
  mon:
```

```
    mon:
      count: 3

      allowMultiplePerNode: false
      volumeClaimTemplate:
        spec:
          storageClassName: managed-premium
          resources:
            requests:
              storage: 10Gi
    cephVersion:
      image: ceph/ceph:v14.2.4-20190917
      allowUnsupported: false
    dashboard:
      enabled: true
      ssl: true
    network:
      hostNetwork: false
    storage:
      storageClassDeviceSets:
      - name: set1
        # The number of OSDs to create from this device set
        count: 4
        # IMPORTANT: If volumes specified by the storageClassName
        # this needs to be set to false. For example, if using th
        # this should be false.
        portable: true
        # Since the OSDs could end up on any node, an effort need
        # across nodes as much as possible. Unfortunately the poo
        # as soon as you have more than one OSD per node. If you
        # choose to schedule many of them on the same node. What
        # Spread Constraints, which is alpha in K8s 1.16. This me
        # enabled for this feature, and Rook also still needs to
        # Another approach for a small number of OSDs is to creat
        # zone (or other set of nodes with a common label) so tha
        # nodes. This would require adding nodeAffinity to the pl
```

```
    placement:

      tolerations:
      - key: storage-node
        operator: Exists
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: agentpool
              operator: In
              values:
              - npstorage
      podAntiAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          podAffinityTerm:
            labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - rook-ceph-osd
              - key: app
                operator: In
                values:
                - rook-ceph-osd-prepare
            topologyKey: kubernetes.io/hostname
    resources:
      limits:
        cpu: "500m"
        memory: "4Gi"
      requests:
        cpu: "500m"
        memory: "2Gi"
```

```
      volumeClaimTemplates:

      - metadata:
          name: data
        spec:
          resources:
            requests:
              storage: 100Gi
          storageClassName: managed-premium
          volumeMode: Block
          accessModes:
            - ReadWriteOnce
  disruptionManagement:
    managePodBudgets: false
    osdMaintenanceTimeout: 30
    manageMachineDisruptionBudgets: false
    machineDisruptionBudgetNamespace: openshift-machine-api
```

So, after a few minutes, you will see some pods running in the *rook-ceph* namespace. Make sure, that the OSD pods a running, before continuing with configuring the storage pool.

```
$ kubectl get pods -n rook-ceph
NAME
csi-cephfsplugin-4qxsv
csi-cephfsplugin-d2klt
csi-cephfsplugin-jps5r
csi-cephfsplugin-kzgrt
csi-cephfsplugin-provisioner-dd9775cd6-nsn8q
csi-cephfsplugin-provisioner-dd9775cd6-tj826
csi-cephfsplugin-rt6x2
csi-cephfsplugin-tdhg6
```

```
csi-rbdplugin-6jkx5
csi-rbdplugin-clfbj

csi-rbdplugin-dxt74
csi-rbdplugin-gspqc
csi-rbdplugin-pfrm4
csi-rbdplugin-provisioner-6dfd6db488-2mrbv
csi-rbdplugin-provisioner-6dfd6db488-2v76h
csi-rbdplugin-qfndk
rook-ceph-crashcollector-aks-npstandard-33852324-vmss00000c8gdp
rook-ceph-crashcollector-aks-npstandard-33852324-vmss00000tfk2s
rook-ceph-crashcollector-aks-npstandard-33852324-vmss00000xfnhx
rook-ceph-crashcollector-aks-npstorage-33852324-vmss000001c6cbc
rook-ceph-crashcollector-aks-npstorage-33852324-vmss000002t6sgc
rook-ceph-mgr-a-5fb458578-s2lgc
rook-ceph-mon-a-7f9fc6f497-mm54j
rook-ceph-mon-b-5dc55c8668-mb976
rook-ceph-mon-d-b7959cf76-txxdt
rook-ceph-operator-5cbdd65df7-htlm7
rook-ceph-osd-0-dd74f9b46-5z2t6
rook-ceph-osd-1-5bcbb6d947-pm5xh
rook-ceph-osd-2-9599bd965-hprb5
rook-ceph-osd-3-557879bf79-8wbjd
rook-ceph-osd-prepare-set1-0-data-sv78n-v969p
rook-ceph-osd-prepare-set1-1-data-r6d46-t2c4q
rook-ceph-osd-prepare-set1-2-data-fl8zq-rrl4r
rook-ceph-osd-prepare-set1-3-data-qrrvf-jjv5b
```

Configuring Storage

Before Rook can provision persistent volumes, either a filesystem or a storage pool should be configured. In our example, a **Ceph Block Pool** is used:

```
apiVersion: ceph.rook.io/v1
kind: CephBlockPool
metadata:
  name: replicapool
  namespace: rook-ceph
spec:
  failureDomain: host
  replicated:
    size: 3
```

Next, we also need a storage class that will be using the Rook cluster / storage pool. In our example, we will not be using Flex Volume (which will be deprecated in furture versions of Rook/Ceph), instead we use **Container Storage Interface**.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: rook-ceph-block
provisioner: rook-ceph.rbd.csi.ceph.com
parameters:
    clusterID: rook-ceph
    pool: replicapool
    imageFormat: "2"
    imageFeatures: layering
    csi.storage.k8s.io/provisioner-secret-name: rook-csi-rbd-pr
    csi.storage.k8s.io/provisioner-secret-namespace: rook-ceph
    csi.storage.k8s.io/node-stage-secret-name: rook-csi-rbd-noo
    csi.storage.k8s.io/node-stage-secret-namespace: rook-ceph
```

```
    csi.storage.k8s.io/fstype: xfs
reclaimPolicy: Delete
```

Test

Now, let's have a look at the dashboard which was also installed when we created the Rook cluster. Therefore, we are port-forwarding the dashboard service to our local machine. The service itself is secured by username/password. The default username is *admin* and the password is stored in a K8s secret. To get the password, simply run the following command.

```
$ kubectl -n rook-ceph get secret rook-ceph-dashboard-password
    -o jsonpath="{['data']['password']}" | base64 --decode && e
# copy the password

$ kubectl port-forward svc/rook-ceph-mgr-dashboard 8443:8443 \
    -n rook-ceph
```

Now access the dasboard by heading to https://localhost:8443/#/dashboard

Ceph Dashboard

As you can see, everything looks healthy. Now let's create a pod that's using a newly created PVC leveraging that Ceph storage class.

PVC

```
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim

metadata:
  name: ceph-pv-claim
spec:
  storageClassName: rook-ceph-block
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ceph-pv-pod
spec:
  volumes:
    - name: ceph-pv-claim
      persistentVolumeClaim:
        claimName: ceph-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: ceph-pv-claim
```

As a result, you will now have an NGINX pod running in your Kuberntes cluster with a PV attached/mounted under /usr/share/nginx/html.

Wrap Up

So…what exactly did we achieve with this solution now? We have created a Ceph storage cluster on an AKS that uses PVCs to manage storage. Okay, so what? Well, the usage of volume mounts in your deployments with Ceph is now **super-fast and rock-solid** , because we do not have to attach physical disks to our worker nodes anymore. We just use the ones we have created during Rook cluster provisioning (remember these four 100GB disks?)! We minimized the amount of “physical attach/detach” actions on our nodes. That’s why now, you won’t see these popular “_WaitForAttach”- or “Can not find LUN for disk”-_errors anymore.

Hope this helps someone out there! Have fun with it.

Update: Benchmarks

Short update on this. Today, I did some benchmarking with *dbench* (https://github.com/leeliu/dbench/) comparing Rook Ceph and “plain” PVCs with the same Azure Premium SSD disks (default AKS StorageClass *managed-premium*, VM types: Standard_DS2_v2). Here are the results…as you can see, it depends on your workload…so, judge by yourself.

Rook Ceph

==========

= Dbench Summary =

Random Read/Write IOPS: 10.6k/571. BW: 107MiB/s / 21.2MiB/s

Average Latency (usec) Read/Write: 715.53/31.70

Sequential Read/Write: 100MiB/s / 43.2MiB/s

Mixed Random Read/Write IOPS: 1651/547

PVC with Azure Premium SSD

> 100GB disk used to have a fair comparison

==========

= Dbench Summary =

Random Read/Write IOPS: 8155/505. BW: 63.7MiB/s / 63.9MiB/s

Average Latency (usec) Read/Write: 505.73/

Sequential Read/Write: 63.6MiB/s / 65.3MiB/s

Mixed Random Read/Write IOPS: 1517/505

Discussion (3)

Subscribe

Add to the discussion

porrascarlos802018 • Nov 6 '20 • Edited on Nov 6

I think this method its great but requires some update to have the PODs running .

a peer Sergio Turrent updated the cluster.yaml to get it working as best effort.

as for 11/06/2020 Our cluster.yaml proposal its as follows:

apiVersion: ceph.rook.io/v1
kind: CephCluster
metadata:
name: rook-ceph
namespace: rook-ceph
spec:
dataDirHostPath: /var/lib/rook
mon:
count: 3
allowMultiplePerNode: false
volumeClaimTemplate:
spec:
storageClassName: managed-premium
resources:
requests:
storage: 10Gi
cephVersion:

image: ceph/ceph:v15.2.4
allowUnsupported: false

dashboard:
enabled: true
ssl: true
network:
hostNetwork: false
placement:
mon:
tolerations:
- key: storage-node
operator: Exists
storage:
storageClassDeviceSets:
- name: set1
The number of OSDs to create from this device set
count: 4
IMPORTANT: If volumes specified by the storageClassName are not portable across nodes
this needs to be set to false. For example, if using the local storage provisioner
this should be false.
portable: true
Since the OSDs could end up on any node, an effort needs to be made to spread the OSDs
across nodes as much as possible. Unfortunately the pod anti-affinity breaks down
as soon as you have more than one OSD per node. If you have more OSDs than nodes, K8s may
choose to schedule many of them on the same node. What we need is the Pod Topology

```
# Spread Constraints, which is alpha in K8s 1.16. This means that a feature gate must be
# enabled for this feature, and Rook also still needs to add support for this feature.
# Another approach for a small number of OSDs is to create a separate device set for each
# zone (or other set of nodes with a common label) so that the OSDs will end up on different
# nodes. This would require adding nodeAffinity to the placement here.
placement:
tolerations:
- key: storage-node
operator: Exists
nodeAffinity:
requiredDuringSchedulingIgnoredDuringExecution:
nodeSelectorTerms:
- matchExpressions:
- key: agentpool
operator: In
values:
- npstorage
podAntiAffinity:
preferredDuringSchedulingIgnoredDuringExecution:
- weight: 100
podAffinityTerm:
labelSelector:
matchExpressions:
- key: app
operator: In
values:
```

- rook-ceph-osd
- key: app
operator: In
values:
- rook-ceph-osd-prepare
topologyKey: kubernetes.io/hostname
resources:
limits:
cpu: "500m"
memory: "4Gi"
requests:
cpu: "500m"
memory: "2Gi"
volumeClaimTemplates:
- metadata:
name: data
spec:
resources:
requests:
storage: 100Gi
storageClassName: managed-premium
volumeMode: Block
accessModes:
- ReadWriteOnce
disruptionManagement:
managePodBudgets: false
osdMaintenanceTimeout: 30
manageMachineDisruptionBudgets: false
machineDisruptionBudgetNamespace: openshift-machine-api

porrascarlos802018 • Nov 6 '20

and just to summarize, step by step procedure I used to get to the point of having the pods running is:

Step 1 creating a nodepool in AKS:
az aks nodepool add --cluster-name aks3del --name npstorage --node-count 2 --resource-group aks3del --node-taints storage-node=true:NoSchedule

Step 2
az aks get-credentials --resource-group aks3del --name aks3del

step 4
kubectl get nodes

Step 5
execute this command:
git clone github.com/rook/rook.git

Step 6
if you are in /home/user , then a new folder called rook should be there.
cd rook

step 7
switch to the /cluster/examples/kubernetes/ceph directory and follow the steps below.

step 8
run the command , kubectl apply -f common.yaml

from the cluster/examples/kubernetes/ceph directory.

step 9
create a new operator.yaml file, do not use the one, from the directory, create a new one and apply it.
vi operator2.yaml
kubectl apply -f operator2.yaml

step 9 validate you have storage class for premiun SSD disks.
kubectl get storageclass

step 10
Create a new cluster2.yaml file, copy and paste the one attached
apply the yaml
kubectl apply -f cluster2.yaml

step 11
validate you got the OSD pods created.
kubectl get pods -n rook-ceph
you will see some pods in init, wait a while
they will eventually start.

rook-ceph-osd will mount a disk using a pvc each one
rook-ceph-osd runs 1 per node.
if a node goes down , the disk dies, the other nodes will have enough data to restore in a new node.
rook-ceph-mon-X are the ones who control the logic on which side it has to replicate the data to have redundancy.
when accessing the data mons are the ones who informs the "client" to know where to retrieve the data.

N SIDDHARTH • Nov 13 '20 • Edited on Nov 13

Your post is very helpful. Do you have a sample document/example to show the working of Ceph NFS CRD in AKS?

Like Reply

Code of Conduct • Report abuse

Read next

L'automatisation dans Azure Kubernetes Services avec Azure DevOps

Thomas Rannou - Feb 10

Horizontal scaling WebSockets on Kubernetes and Node.js

Tarek N. Elsamni - Feb 9

Understanding Kubernetes in a visual way (in 🎥 video): part 1 – Pods

Aurélie Vache - Feb 6

Kubernetes - Delete multiple resources

Maxime Guilbert - Feb 9