Rook 1.4 Document

Rook 是一个开源的cloud-native storage编排, 提供平台和框架；为各种存储解决方案提供平台、框架和支持，以便与云原生环境本地集成。 Rook 将存储软件转变为自我管理、自我扩展和自我修复的存储服务，它通过自动…

目　录

致谢

当前文档 《Rook 1.4 Document》 由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2020-08-22。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：rook https://rook.io/docs/rook/v1.3/

文档地址：http://www.bookstack.cn/books/rook-1.4-en

书栈官网：https://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Rook

Rook is an open source **cloud-native storage orchestrator**, providing the platform, framework, and support for a diverse set of storage solutions to natively integrate with cloud-native environments.

Rook turns storage software into self-managing, self-scaling, and self-healing storage services. It does this by automating deployment, bootstrapping, configuration, provisioning, scaling, upgrading, migration, disaster recovery, monitoring, and resource management. Rook uses the facilities provided by the underlying cloud-native container management, scheduling and orchestration platform to perform its duties.

Rook integrates deeply into cloud native environments leveraging extension points and providing a seamless experience for scheduling, lifecycle management, resource management, security, monitoring, and user experience.

For more details about the status of storage solutions currently supported by Rook, please refer to the project status section of the Rook repository. We plan to continue adding support for other storage systems and environments based on community demand and engagement in future releases.

Quick Start Guides

Starting Rook in your cluster is as simple as a few `kubectl` commands depending on the storage provider. See our Quickstart guide list for the detailed instructions for each storage provider.

Storage Provider Designs

High-level Storage Provider design documents:

Storage Provider	Status	Description
Ceph	Stable	Ceph is a highly scalable distributed storage solution for block storage, object storage, and shared filesystems with years of production deployments.
EdgeFS	Stable	EdgeFS is high-performance and fault-tolerant decentralized data fabric with access to object, file, NoSQL and block.

Low level design documentation for supported list of storage systems collected at design docs section.

Need help? Be sure to join the Rook Slack

If you have any questions along the way, please don’t hesitate to ask us in our Slack

channel. You can sign up for our Slack here.

Quickstart Guides

Welcome to Rook! We hope you have a great experience installing the Rook **cloud-native storage orchestrator** platform to enable highly available, durable storage in your Kubernetes cluster.

If you have any questions along the way, please don't hesitate to ask us in our Slack channel. You can sign up for our Slack here.

Rook provides a growing number of storage providers to a Kubernetes cluster, each with its own operator to deploy and manage the resources for the storage provider.

Follow these guides to get started with each provider:

Storage Provider	Status	Description
Ceph	Stable / V1	Ceph is a highly scalable distributed storage solution for block storage, object storage, and shared filesystems with years of production deployments.
EdgeFS	Stable / V1	EdgeFS is high-performance and fault-tolerant decentralized data fabric with access to object, file, NoSQL and block.
Cassandra	Alpha	Cassandra is a highly available NoSQL database featuring lightning fast performance, tunable consistency and massive scalability.
CockroachDB	Alpha	CockroachDB is a cloud-native SQL database for building global, scalable cloud services that survive disasters.
NFS	Alpha	NFS allows remote hosts to mount filesystems over a network and interact with those filesystems as though they are mounted locally.
YugabyteDB	Alpha	YugaByteDB is a high-performance, cloud-native distributed SQL database which can tolerate disk, node, zone and region failures automatically.

Cassandra Quickstart

Cassandra is a highly available, fault tolerant, peer-to-peer NoSQL database featuring lightning fast performance and tunable consistency. It provides massive scalability with no single point of failure.

Scylla is a close-to-the-hardware rewrite of Cassandra in C++. It features a shared nothing architecture that enables true linear scaling and major hardware optimizations that achieve ultra-low latencies and extreme throughput. It is a drop-in replacement for Cassandra and uses the same interfaces, so it is also supported by Rook.

Prerequisites

A Kubernetes cluster is necessary to run the Rook Cassandra operator. To make sure you have a Kubernetes cluster that is ready for `Rook`, you can follow these instructions (the flexvolume plugin is not necessary for Cassandra)

Deploy Cassandra Operator

First deploy the Rook Cassandra Operator using the following commands:

```
1. git clone --single-branch --branch release-1.4 https://github.com/rook/rook.git
2. cd rook/cluster/examples/kubernetes/cassandra
3. kubectl apply -f operator.yaml
```

This will install the operator in namespace rook-cassandra-system. You can check if the operator is up and running with:

```
1. kubectl -n rook-cassandra-system get pod
```

Create and Initialize a Cassandra/Scylla Cluster

Now that the operator is running, we can create an instance of a Cassandra/Scylla cluster by creating an instance of the `clusters.cassandra.rook.io` resource. Some of that resource's values are configurable, so feel free to browse `cluster.yaml` and tweak the settings to your liking. Full details for all the configuration options can be found in the Cassandra Cluster CRD documentation.

When you are ready to create a Cassandra cluster, simply run:

```
1. kubectl create -f cluster.yaml
```

We can verify that a Kubernetes object has been created that represents our new Cassandra cluster with the command below. This is important because it shows that Rook has successfully extended Kubernetes to make Cassandra clusters a first class citizen in the Kubernetes cloud-native environment.

```
1. kubectl -n rook-cassandra get clusters.cassandra.rook.io
```

To check if all the desired members are running, you should see the same number of entries from the following command as the number of members that was specified in `cluster.yaml` :

```
1. kubectl -n rook-cassandra get pod -l app=rook-cassandra
```

You can also track the state of a Cassandra cluster from its status. To check the current status of a Cluster, run:

```
1. kubectl -n rook-cassandra describe clusters.cassandra.rook.io rook-cassandra
```

Accessing the Database

- From kubectl:

To get a `cqlsh` shell in your new Cluster:

```
1. kubectl exec -n rook-cassandra -it rook-cassandra-east-1-east-1a-0 -- cqlsh
2. > DESCRIBE KEYSPACES;
```

- From inside a Pod:

When you create a new Cluster, Rook automatically creates a Service for the clients to use in order to access the Cluster. The service’s name follows the convention `<cluster-name>-client` . You can see this Service in you cluster by running:

```
1. kubectl -n rook-cassandra describe service rook-cassandra-client
```

Pods running inside the Kubernetes cluster can use this Service to connect to Cassandra. Here’s an example using the Python Driver:

```
1. from cassandra.cluster import Cluster
2.
3. cluster = Cluster(['rook-cassandra-client.rook-cassandra.svc.cluster.local'])
4. session = cluster.connect()
```

Scale Up

The operator supports scale up of a rack as well as addition of new racks. To make the changes, you can use:

```
1. kubectl edit clusters.cassandra.rook.io rook-cassandra
```

- To scale up a rack, change the `Spec.Members` field of the rack to the desired value.
- To add a new rack, append the `racks` list with a new rack. Remember to choose a different rack name for the new rack.
- After editing and saving the yaml, check your cluster's Status and Events for information on what's happening:

```
1. kubectl -n rook-cassandra describe clusters.cassandra.rook.io rook-cassandra
```

Scale Down

The operator supports scale down of a rack. To make the changes, you can use:

```
1. kubectl edit clusters.cassandra.rook.io rook-cassandra
```

- To scale down a rack, change the `Spec.Members` field of the rack to the desired value.
- After editing and saving the yaml, check your cluster's Status and Events for information on what's happening:

```
1. kubectl -n rook-cassandra describe clusters.cassandra.rook.io rook-cassandra
```

Clean Up

To clean up all resources associated with this walk-through, you can run the commands below.

> **NOTE**: that this will destroy your database and delete all of its associated data.

```
1. kubectl delete -f cluster.yaml
2. kubectl delete -f operator.yaml
```

Troubleshooting

If the cluster does not come up, the first step would be to examine the operator's logs:

```
1. kubectl -n rook-cassandra-system logs -l app=rook-cassandra-operator
```

If everything looks OK in the operator logs, you can also look in the logs for one of the Cassandra instances:

```
kubectl -n rook-cassandra logs rook-cassandra-0
```

Cassandra Monitoring

To enable jmx_exporter for cassandra rack, you should specify `jmxExporterConfigMapName` option for rack in CassandraCluster CRD.

For example:

```
apiVersion: cassandra.rook.io/v1alpha1
kind: Cluster
metadata:
  name: my-cassandra
  namespace: rook-cassandra
spec:
  ...
  datacenter:
    name: my-datacenter
    racks:
    - name: my-rack
      members: 3
      jmxExporterConfigMapName: jmx-exporter-settings
      storage:
        volumeClaimTemplates:
        - metadata:
            name: rook-cassandra-data
          spec:
            storageClassName: my-storage-class
            resources:
              requests:
                storage: 200Gi
```

Simple config map example to get all metrics:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: jmx-exporter-settings
  namespace: rook-cassandra
data:
  jmx_exporter_config.yaml: |
    lowercaseOutputLabelNames: true
    lowercaseOutputName: true
    whitelistObjectNames: ["org.apache.cassandra.metrics:*"]
```

ConfigMap’s data field must contain `jmx_exporter_config.yaml` key with jmx exporter

settings.

There is no automatic reloading mechanism for pods when the config map updated. After the configmap changed, you should restart all rack pods manually:

```
1. NAMESPACE=<namespace>
2. CLUSTER=<cluster_name>
3. RACKS=$(kubectl get sts -n ${NAMESPACE} -l "cassandra.rook.io/cluster=${CLUSTER}")
4. echo ${RACKS} | xargs -n1 kubectl rollout restart -n ${NAMESPACE}
```

Ceph Storage Quickstart

This guide will walk you through the basic setup of a Ceph cluster and enable you to consume block, object, and file storage from other pods running in your cluster.

Minimum Version

Kubernetes **v1.11** or higher is supported by Rook.

Prerequisites

To make sure you have a Kubernetes cluster that is ready for `Rook`, you can follow these instructions.

In order to configure the Ceph storage cluster, at least one of these local storage options are required:

- Raw devices (no partitions or formatted filesystems)
- Raw partitions (no formatted filesystem)
- PVs available from a storage class in `block` mode

You can confirm whether your partitions or devices are formatted filesystems with the following command.

```
lsblk -f
NAME                  FSTYPE      LABEL UUID                                   MOUNTPOINT
vda
└─vda1                LVM2_member       eSO50t-GkUV-YKTH-WsGq-hNJY-eKNf-3i07IB
  ├─ubuntu--vg-root   ext4              c2366f76-6e21-4f10-a8f3-6776212e2fe4   /
  └─ubuntu--vg-swap_1 swap              9492a3dc-ad75-47cd-9596-678e8cf17ff9   [SWAP]
vdb
```

If the `FSTYPE` field is not empty, there is a filesystem on top of the corresponding device. In this case, you can use vdb for Ceph and can't use vda and its partitions.

TL;DR

If you're feeling lucky, a simple Rook cluster can be created with the following kubectl commands and example yaml files. For the more detailed install, skip to the next section to deploy the Rook operator.

```
git clone --single-branch --branch release-1.4 https://github.com/rook/rook.git
cd rook/cluster/examples/kubernetes/ceph
kubectl create -f common.yaml
kubectl create -f operator.yaml
```

```
kubectl create -f cluster.yaml
```

After the cluster is running, you can create block, object, or file storage to be consumed by other applications in your cluster.

Cluster Environments

The Rook documentation is focused around starting Rook in a production environment. Examples are also provided to relax some settings for test environments. When creating the cluster later in this guide, consider these example cluster manifests:

- cluster.yaml: Cluster settings for a production cluster running on bare metal
- cluster-on-pvc.yaml: Cluster settings for a production cluster running in a dynamic cloud environment
- cluster-test.yaml: Cluster settings for a test environment such as minikube.

See the Ceph examples for more details.

Deploy the Rook Operator

The first step is to deploy the Rook operator. Check that you are using the example yaml files that correspond to your release of Rook. For more options, see the examples documentation.

```
cd cluster/examples/kubernetes/ceph
kubectl create -f common.yaml
kubectl create -f operator.yaml

## verify the rook-ceph-operator is in the `Running` state before proceeding
kubectl -n rook-ceph get pod
```

You can also deploy the operator with the Rook Helm Chart.

Create a Rook Ceph Cluster

Now that the Rook operator is running we can create the Ceph cluster. For the cluster to survive reboots, make sure you set the `dataDirHostPath` property that is valid for your hosts. For more settings, see the documentation on configuring the cluster.

Create the cluster:

```
kubectl create -f cluster.yaml
```

Use `kubectl` to list pods in the `rook-ceph` namespace. You should be able to see the following pods once they are all running. The number of osd pods will depend on the number of nodes in the cluster and the number of devices configured. If you did not

modify the `cluster.yaml` above, it is expected that one OSD will be created per node. The CSI, `rook-ceph-agent`, and `rook-discover` pods are also optional depending on your settings.

> If the `rook-ceph-mon`, `rook-ceph-mgr`, or `rook-ceph-osd` pods are not created, please refer to the Ceph common issues for more details and potential solutions.

```
$ kubectl -n rook-ceph get pod
NAME                                            READY   STATUS      RESTARTS   AGE
csi-cephfsplugin-provisioner-d77bb49c6-n5tgs    5/5     Running     0          140s
csi-cephfsplugin-provisioner-d77bb49c6-v9rvn    5/5     Running     0          140s
csi-cephfsplugin-rthrp                          3/3     Running     0          140s
csi-rbdplugin-hbsm7                             3/3     Running     0          140s
csi-rbdplugin-provisioner-5b5cd64fd-nvk6c       6/6     Running     0          140s
csi-rbdplugin-provisioner-5b5cd64fd-q7bxl       6/6     Running     0          140s
rook-ceph-agent-4zkg8                           1/1     Running     0          140s
rook-ceph-crashcollector-minikube-5b57b7c5d4-hfldl   1/1     Running     0          105s
rook-ceph-mgr-a-64cd7cdf54-j8b5p                1/1     Running     0          77s
rook-ceph-mon-a-694bb7987d-fp9w7                1/1     Running     0          105s
rook-ceph-mon-b-856fdd5cb9-5h2qk                1/1     Running     0          94s
rook-ceph-mon-c-57545897fc-j576h                1/1     Running     0          85s
rook-ceph-operator-85f5b946bd-s8grz             1/1     Running     0          92m
rook-ceph-osd-0-6bb747b6c5-lnvb6                1/1     Running     0          23s
rook-ceph-osd-1-7f67f9646d-44p7v                1/1     Running     0          24s
rook-ceph-osd-2-6cd4b776ff-v4d68                1/1     Running     0          25s
rook-ceph-osd-prepare-node1-vx2rz               0/2     Completed   0          60s
rook-ceph-osd-prepare-node2-ab3fd               0/2     Completed   0          60s
rook-ceph-osd-prepare-node3-w4xyz               0/2     Completed   0          60s
rook-discover-dhkb8                             1/1     Running     0          140s
```

To verify that the cluster is in a healthy state, connect to the Rook toolbox and run the `ceph status` command.

- All mons should be in quorum
- A mgr should be active
- At least one OSD should be active
- If the health is not `HEALTH_OK`, the warnings or errors should be investigated

```
$ ceph status
  cluster:
    id:     a0452c76-30d9-4c1a-a948-5d8405f19a7c
    health: HEALTH_OK

  services:
    mon: 3 daemons, quorum a,b,c (age 3m)
    mgr: a(active, since 2m)
    osd: 3 osds: 3 up (since 1m), 3 in (since 1m)
...
```

If the cluster is not healthy, please refer to the Ceph common issues for more details

and potential solutions.

Storage

For a walkthrough of the three types of storage exposed by Rook, see the guides for:

- **Block**: Create block storage to be consumed by a pod
- **Object**: Create an object store that is accessible inside or outside the Kubernetes cluster
- **Shared Filesystem**: Create a filesystem to be shared across multiple pods

Ceph Dashboard

Ceph has a dashboard in which you can view the status of your cluster. Please see the dashboard guide for more details.

Tools

We have created a toolbox container that contains the full suite of Ceph clients for debugging and troubleshooting your Rook cluster. Please see the toolbox readme for setup and usage information. Also see our advanced configuration document for helpful maintenance and tuning examples.

Monitoring

Each Rook cluster has some built in metrics collectors/exporters for monitoring with Prometheus. To learn how to set up monitoring for your Rook cluster, you can follow the steps in the monitoring guide.

Teardown

When you are done with the test cluster, see these instructions to clean up the cluster.

CockroachDB Quickstart

CockroachDB is a cloud-native SQL database for building global, scalable cloud services that survive disasters. Rook provides an operator to deploy and manage CockroachDB clusters.

Prerequisites

A Kubernetes cluster is necessary to run the Rook CockroachDB operator. To make sure you have a Kubernetes cluster that is ready for `Rook`, you can follow these instructions.

Deploy CockroachDB Operator

First deploy the Rook CockroachDB operator using the following commands:

```
1. git clone --single-branch --branch release-1.4 https://github.com/rook/rook.git
2. cd rook/cluster/examples/kubernetes/cockroachdb
3. kubectl create -f operator.yaml
```

You can check if the operator is up and running with:

```
1.  kubectl -n rook-cockroachdb-system get pod
```

Create and Initialize CockroachDB Cluster

Now that the operator is running, we can create an instance of a CockroachDB cluster by creating an instance of the `cluster.cockroachdb.rook.io` resource. Some of that resource's values are configurable, so feel free to browse `cluster.yaml` and tweak the settings to your liking. Full details for all the configuration options can be found in the CockroachDB Cluster CRD documentation.

When you are ready to create a CockroachDB cluster, simply run:

```
1. kubectl create -f cluster.yaml
```

We can verify that a Kubernetes object has been created that represents our new CockroachDB cluster with the command below. This is important because it shows that Rook has successfully extended Kubernetes to make CockroachDB clusters a first class citizen in the Kubernetes cloud-native environment.

```
1. kubectl -n rook-cockroachdb get clusters.cockroachdb.rook.io
```

To check if all the desired replicas are running, you should see the same number of entries from the following command as the replica count that was specified in `cluster.yaml` :

```
kubectl -n rook-cockroachdb get pod -l app=rook-cockroachdb
```

Accessing the Database

To use the `cockroach sql` client to connect to the database cluster, run the following command in its entirety:

```
kubectl -n rook-cockroachdb-system exec -it $(kubectl -n rook-cockroachdb-system get pod -l app=rook-cockroachdb-operator -o jsonpath='{.items[0].metadata.name}') -- /cockroach/cockroach sql --insecure --host=cockroachdb-public.rook-cockroachdb
```

This will land you in a prompt where you can begin to run SQL commands directly on the database cluster.

Example:

```
root@cockroachdb-public.rook-cockroachdb:26257/> show databases;
+----------+
| Database |
+----------+
| system   |
| test     |
+----------+
(2 rows)

Time: 2.105065ms
```

Example App

If you want to run an example application to exercise your new CockroachDB cluster, there is a load generator application in the same directory as the operator and cluster resource files. The load generator will start writing random key-value pairs to the database cluster, verifying that the cluster is functional and can handle reads and writes.

The rate at which the load generator writes data is configurable, so feel free to tweak the values in `loadgen-kv.yaml` . Setting `--max-rate=0` will enable the load generator to go as fast as it can, putting a large amount of load onto your database cluster.

To run the load generator example app, simply run:

```
kubectl create -f loadgen-kv.yaml
```

You can check on the progress and statistics of the load generator by running:

```
kubectl -n rook-cockroachdb logs -l app=loadgen
```

To connect to the database and view the data that the load generator has written, run the following command:

```
kubectl -n rook-cockroachdb-system exec -it $(kubectl -n rook-cockroachdb-system get pod -l app=rook-cockroachdb-operator -o jsonpath='{.items[0].metadata.name}') -- /cockroach/cockroach sql --insecure --host=cockroachdb-public.rook-cockroachdb -d test -e 'select * from kv'
```

Clean up

To clean up all resources associated with this walk-through, you can run the commands below.

> **NOTE**: that this will destroy your database and delete all of its associated data.

```
kubectl delete -f loadgen-kv.yaml
kubectl delete -f cluster.yaml
kubectl delete -f operator.yaml
```

Troubleshooting

If the cluster does not come up, the first step would be to examine the operator's logs:

```
kubectl -n rook-cockroachdb-system logs -l app=rook-cockroachdb-operator
```

If everything looks OK in the operator logs, you can also look in the logs for one of the CockroachDB instances:

```
kubectl -n rook-cockroachdb logs rook-cockroachdb-0
```

EdgeFS Data Fabric Quickstart

EdgeFS Data Fabric virtualzing common storage protocols and enables multi-cluster, multi-region data flow topologies.

This guide will walk you through the basic setup of a EdgeFS cluster namespaces and enable you to consume S3 object, NFS/SMB file access, and iSCSI block storage from other pods running in your cluster, in decentralized ways.

Minimum Version

EdgeFS operator, CSI plugin and CRDs were tested with Kubernetes **v1.13** or higher.

Prerequisites

To make sure you have a Kubernetes cluster that is ready for `Rook`, you can follow these instructions.

A minimum of 3 storage devices are required with the default data replication count of 3. (To test EdgeFS on a single-node cluster with only one device or storage directory, set `sysRepCount` to `1` under the `rook-edgefs` `Cluster` object manifest in `cluster/examples/kubernetes/edgefs/cluster.yml`)

To operate efficiently EdgeFS requires 1 CPU core and 1GB of memory per storage device. Minimal memory requirement for EdgeFS target pod is 4GB. To get maximum out of SSD/NVMe device we recommend to double requirements to 2 CPU and 2GB per device.

If you are using `dataDirHostPath` to persist rook data on kubernetes hosts, make sure your host has at least 5GB of space available on the specified path.

We recommend you to configure EdgeFS to use of raw devices and equal distribution of available storage capacity.

> **IMPORTANT**: EdgeFS will automatically adjust deployment nodes to use larger then 128KB data chunks, with the following addition to /etc/sysctl.conf:

```
net.core.rmem_default = 80331648
net.core.rmem_max = 80331648
net.core.wmem_default = 33554432
net.core.wmem_max = 50331648
vm.dirty_ratio = 10
vm.dirty_background_ratio = 5
vm.swappiness = 15
```

To turn off this node adjustment need to enable `skipHostPrepare` option in cluster CRD configuring the cluster.

TL;DR

If you're feeling lucky, a simple EdgeFS Rook cluster can be created with the following kubectl commands. For the more detailed install, skip to the next section to deploy the Rook operator.

```
1. git clone --single-branch --branch release-1.4 https://github.com/rook/rook.git
2. cd rook/cluster/examples/kubernetes/edgefs
3. kubectl create -f operator.yaml
4. kubectl create -f cluster.yaml
```

After the cluster is running, you can create NFS, SMB, S3, or iSCSI storage to be consumed by other applications in your cluster.

Deploy the Rook Operator

The first step is to deploy the Rook system components, which include the Rook agent running on each node in your cluster as well as Rook operator pod.

Note that Google Cloud users need to explicitly grant user permission to create roles:

```
1. kubectl create clusterrolebinding cluster-admin-binding --clusterrole cluster-admin --user $(gcloud config get-value account)
```

Now you ready to create operator:

```
1. cd cluster/examples/kubernetes/edgefs
2. kubectl create -f operator.yaml
3.
4. # verify the rook-edgefs-operator, and rook-discover pods are in the `Running` state before proceeding
5. kubectl -n rook-edgefs-system get pod
```

Create a Rook Cluster

Now that the Rook operator, and discover pods are running, we can create the Rook cluster. For the cluster to survive reboots, make sure you set the `dataDirHostPath` property. For more settings, see the documentation on configuring the cluster.

Edit the cluster spec in `cluster.yaml` file.

Create the cluster:

```
1. kubectl create -f cluster.yaml
```

Use `kubectl` to list pods in the `rook-edgefs` namespace. You should be able to see the following pods once they are all running. The number of target pods will depend on the

number of nodes in the cluster and the number of devices and directories configured.

```
$ kubectl -n rook-edgefs get pod
rook-edgefs          rook-edgefs-mgr-7c76cb564d-56sxb       1/1     Running   0          24s
rook-edgefs          rook-edgefs-target-0                   3/3     Running   0          24s
rook-edgefs          rook-edgefs-target-1                   3/3     Running   0          24s
rook-edgefs          rook-edgefs-target-2                   3/3     Running   0          24s
```

Notice that EdgeFS Targets are running as StatefulSet.

Storage

For a walkthrough of the types of Storage CRDs exposed by EdgeFS Rook, see the guides for:

- **NFS Server**: Create Scale-Out NFS storage to be consumed by multiple pods, simultaneously
- **SMB Server**: Create Scale-Out Microsoft CIFS/SMB storage to be consumed by multiple pods, simultaneously
- **S3X**: Create an Extended S3 HTTP/2 compatible object and key-value store that is accessible inside or outside the Kubernetes cluster
- **AWS S3**: Create an AWS S3 compatible object store that is accessible inside or outside the Kubernetes cluster
- **iSCSI Target**: Create low-latency and high-throughput iSCSI block to be consumed by a pod

CSI Integration

EdgeFS comes with built-in gRPC management services, which are tightly integrated into Kubernetes provisioning and attaching CSI framework. Please see the EdgeFS CSI for setup and usage information.

EdgeFS Dashboard and Monitoring

Each Rook cluster has some built in metrics collectors/exporters for monitoring with Prometheus. In additional to classic monitoring parameters, it has a built-in multi-tenancy, multi-service capability. For instance, you can quickly discover most loaded tenants, buckets or services. To learn how to set up monitoring for your Rook cluster, you can follow the steps in the monitoring guide.

Teardown

When you are done with the cluster, simply delete CRDs in reverse order. You may want to re-format your raw disks with `wipefs -a` command. Or if you using raw devices and want to keep same storage configuration but change some resource or networking

parameters, consider to use `devicesResurrectMode` .

Network Filesystem (NFS)

NFS allows remote hosts to mount filesystems over a network and interact with those filesystems as though they are mounted locally. This enables system administrators to consolidate resources onto centralized servers on the network.

Prerequisites

1. A Kubernetes cluster is necessary to run the Rook NFS operator. To make sure you have a Kubernetes cluster that is ready for `Rook`, you can follow these instructions.
2. The desired volume to export needs to be attached to the NFS server pod via a PVC. Any type of PVC can be attached and exported, such as Host Path, AWS Elastic Block Store, GCP Persistent Disk, CephFS, Ceph RBD, etc. The limitations of these volumes also apply while they are shared by NFS. You can read further about the details and limitations of these volumes in the Kubernetes docs.
3. NFS client packages must be installed on all nodes where Kubernetes might run pods with NFS mounted. Install `nfs-utils` on CentOS nodes or `nfs-common` on Ubuntu nodes.

Deploy NFS Operator

First deploy the Rook NFS operator using the following commands:

```
git clone --single-branch --branch release-1.4 https://github.com/rook/rook.git
cd rook/cluster/examples/kubernetes/nfs
kubectl create -f common.yaml
kubectl create -f provisioner.yaml
kubectl create -f operator.yaml
```

You can check if the operator is up and running with:

```
kubectl -n rook-nfs-system get pod

NAME                                READY   STATUS    RESTARTS   AGE
rook-nfs-operator-879f5bf8b-gnwht   1/1     Running   0          29m
rook-nfs-provisioner-65f4874c8f-kkz6b   1/1     Running   0          29m
```

Deploy NFS Admission Webhook (Optional)

Admission webhooks are HTTP callbacks that receive admission requests to the API server. Two types of admission webhooks is validating admission webhook and mutating admission webhook. NFS Operator support validating admission webhook which validate

the NFSServer object sent to the API server before stored in the etcd (persisted).

To enable admission webhook on NFS such as validating admission webhook, you need to do as following:

First, ensure that `cert-manager` is installed. If it is not installed yet, you can install it as described in the `cert-manager` installation documentation. Alternatively, you can simply just run the single command below:

```
kubectl apply --validate=false -f https://github.com/jetstack/cert-manager/releases/download/v0.15.1/cert-manager.yaml
```

This will easily get the latest version (`v0.15.1`) of `cert-manager` installed. After that completes, make sure the cert-manager component deployed properly and is in the `Running` status:

```
kubectl get -n cert-manager pod

NAME                                       READY   STATUS    RESTARTS   AGE
cert-manager-7747db9d88-jmw2f              1/1     Running   0          2m1s
cert-manager-cainjector-87c85c6ff-dhtl8    1/1     Running   0          2m1s
cert-manager-webhook-64dc9fff44-5g565      1/1     Running   0          2m1s
```

Once `cert-manager` is running, you can now deploy the NFS webhook:

```
kubectl create -f webhook.yaml
```

Verify the webhook is up and running:

```
kubectl -n rook-nfs-system get pod

NAME                                  READY   STATUS    RESTARTS   AGE
rook-nfs-operator-78d86bf969-k7lqp    1/1     Running   0          102s
rook-nfs-provisioner-7b5ff479f6-688dm 1/1     Running   0          102s
rook-nfs-webhook-74749cbd46-6jw2w     1/1     Running   0          102s
```

Create Openshift Security Context Constraints (Optional)

On OpenShift clusters, we will need to create some additional security context constraints. If you are **not** running in OpenShift you can skip this and go to the next section.

To create the security context constraints for nfs-server pods, we can use the following yaml, which is also found in `scc.yaml` under `/cluster/examples/kubernetes/nfs` .

> *NOTE: Older versions of OpenShift may require* `apiVersion: v1`

```
kind: SecurityContextConstraints
apiVersion: security.openshift.io/v1
metadata:
  name: rook-nfs
allowHostDirVolumePlugin: true
allowHostIPC: false
allowHostNetwork: false
allowHostPID: false
allowHostPorts: false
allowPrivilegedContainer: false
allowedCapabilities:
- SYS_ADMIN
- DAC_READ_SEARCH
defaultAddCapabilities: null
fsGroup:
  type: MustRunAs
priority: null
readOnlyRootFilesystem: false
requiredDropCapabilities:
- KILL
- MKNOD
- SYS_CHROOT
runAsUser:
  type: RunAsAny
seLinuxContext:
  type: MustRunAs
supplementalGroups:
  type: RunAsAny
volumes:
- configMap
- downwardAPI
- emptyDir
- persistentVolumeClaim
- secret
users:
  - system:serviceaccount:rook-nfs:rook-nfs-server
```

You can create scc with following command:

```
oc create -f scc.yaml
```

Create and Initialize NFS Server

Now that the operator is running, we can create an instance of a NFS server by creating an instance of the `nfsservers.nfs.rook.io` resource. The various fields and options of the NFS server resource can be used to configure the server and its volumes to export. Full details of the available configuration options can be found in the NFS CRD documentation.

This guide has 2 main examples that demonstrate exporting volumes with a NFS server:

1. Default StorageClass example
2. Rook Ceph volume example

Default StorageClass example

This first example will walk through creating a NFS server instance that exports storage that is backed by the default `StorageClass` for the environment you happen to be running in. In some environments, this could be a host path, in others it could be a cloud provider virtual disk. Either way, this example requires a default `StorageClass` to exist.

Start by saving the below NFS CRD instance definition to a file called `nfs.yaml`:

```
apiVersion: v1
kind: Namespace
metadata:
  name:  rook-nfs
---
# A default storageclass must be present
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-default-claim
  namespace: rook-nfs
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
---
apiVersion: nfs.rook.io/v1alpha1
kind: NFSServer
metadata:
  name: rook-nfs
  namespace: rook-nfs
spec:
  replicas: 1
  exports:
  - name: share1
    server:
      accessMode: ReadWrite
      squash: "none"
    # A Persistent Volume Claim must be created before creating NFS CRD instance.
    persistentVolumeClaim:
      claimName: nfs-default-claim
  # A key/value list of annotations
  annotations:
    rook: nfs
```

With the `nfs.yaml` file saved, now create the NFS server as shown:

```
kubectl create -f nfs.yaml
```

We can verify that a Kubernetes object has been created that represents our new NFS server and its export with the command below.

```
kubectl -n rook-nfs get nfsservers.nfs.rook.io

NAME       AGE   STATE
rook-nfs   32s   Running
```

Verify that the NFS server pod is up and running:

```
kubectl -n rook-nfs get pod -l app=rook-nfs

NAME         READY     STATUS    RESTARTS   AGE
rook-nfs-0   1/1       Running   0          2m
```

If the NFS server pod is in the `Running` state, then we have successfully created an exported NFS share that clients can start to access over the network.

Accessing the Export

Since Rook version v1.0, Rook supports dynamic provisioning of NFS. This example will be showing how dynamic provisioning feature can be used for nfs.

Once the NFS Operator and an instance of NFSServer is deployed. A storageclass similar to below example has to be created to dynamically provisioning volumes.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  labels:
    app: rook-nfs
  name: rook-nfs-share1
parameters:
  exportName: share1
  nfsServerName: rook-nfs
  nfsServerNamespace: rook-nfs
provisioner: rook.io/nfs-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

You can save it as a file, eg: called `sc.yaml` Then create storageclass with following command.

```
kubectl create -f sc.yaml
```

> **NOTE**: The StorageClass need to have the following 3 parameters passed.
>
> 1. `exportName` : It tells the provisioner which export to use for provisioning the volumes.
> 2. `nfsServerName` : It is the name of the NFSServer instance.
> 3. `nfsServerNamespace` : It namespace where the NFSServer instance is running.

Once the above storageclass has been created create a PV claim referencing the storageclass as shown in the example given below.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: rook-nfs-pv-claim
spec:
  storageClassName: "rook-nfs-share1"
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Mi
```

You can also save it as a file, eg: called `pvc.yaml` Then create PV claim with following command.

```
kubectl create -f pvc.yaml
```

Consuming the Export

Now we can consume the PV that we just created by creating an example web server app that uses the above `PersistentVolumeClaim` to claim the exported volume. There are 2 pods that comprise this example:

1. A web server pod that will read and display the contents of the NFS share
2. A writer pod that will write random data to the NFS share so the website will continually update

Start both the busybox pod (writer) and the web server from the `cluster/examples/kubernetes/nfs` folder:

```
kubectl create -f busybox-rc.yaml
kubectl create -f web-rc.yaml
```

Let’s confirm that the expected busybox writer pod and web server pod are **all** up and in the `Running` state:

```
kubectl get pod -l app=nfs-demo
```

In order to be able to reach the web server over the network, let's create a service for it:

```
kubectl create -f web-service.yaml
```

We can then use the busybox writer pod we launched before to check that nginx is serving the data appropriately. In the below 1-liner command, we use `kubectl exec` to run a command in the busybox writer pod that uses `wget` to retrieve the web page that the web server pod is hosting. As the busybox writer pod continues to write a new timestamp, we should see the returned output also update every ~10 seconds or so.

```
> echo; kubectl exec $(kubectl get pod -l app=nfs-demo,role=busybox -o jsonpath='{.items[0].metadata.name}') -
- wget -qO- http://$(kubectl get services nfs-web -o jsonpath='{.spec.clusterIP}'); echo

Thu Oct 22 19:28:55 UTC 2015
nfs-busybox-w3s4t
```

Rook Ceph volume example

In this alternative example, we will use a different underlying volume as an export for the NFS server. These steps will walk us through exporting a Ceph RBD block volume so that clients can access it across the network.

First, you have to follow these instructions to deploy a sample Rook Ceph cluster that can be attached to the NFS server pod for sharing. After the Rook Ceph cluster is up and running, we can create proceed with creating the NFS server.

Save this PVC and NFS CRD instance as `nfs-ceph.yaml` :

```
apiVersion: v1
kind: Namespace
metadata:
  name:  rook-nfs
---
# A rook ceph cluster must be running
# Create a rook ceph cluster using examples in rook/cluster/examples/kubernetes/ceph
# Refer to https://rook.io/docs/rook/master/ceph-quickstart.html for a quick rook cluster setup
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-ceph-claim
  namespace: rook-nfs
spec:
  storageClassName: rook-ceph-block
  accessModes:
  - ReadWriteMany
  resources:
    requests:
```

```
      storage: 2Gi
---
apiVersion: nfs.rook.io/v1alpha1
kind: NFSServer
metadata:
  name: rook-nfs
  namespace: rook-nfs
spec:
  replicas: 1
  exports:
  - name: nfs-share
    server:
      accessMode: ReadWrite
      squash: "none"
    # A Persistent Volume Claim must be created before creating NFS CRD instance.
    # Create a Ceph cluster for using this example
    # Create a ceph PVC after creating the rook ceph cluster using ceph-pvc.yaml
    persistentVolumeClaim:
      claimName: nfs-ceph-claim
```

Create the NFS server instance that you saved in `nfs-ceph.yaml`:

```
kubectl create -f nfs-ceph.yaml
```

After the NFS server pod is running, follow the same instructions from the previous example to access and consume the NFS share.

Teardown

To clean up all resources associated with this walk-through, you can run the commands below.

```
kubectl delete -f web-service.yaml
kubectl delete -f web-rc.yaml
kubectl delete -f busybox-rc.yaml
kubectl delete -f pvc.yaml
kubectl delete -f pv.yaml
kubectl delete -f nfs.yaml
kubectl delete -f nfs-ceph.yaml
kubectl delete -f operator.yaml
kubectl delete -f provisioner.yaml
kubectl delete -f webhook.yaml # if deployed
kubectl delete -f common.yaml
```

Troubleshooting

If the NFS server pod does not come up, the first step would be to examine the NFS operator's logs:

```
1. kubectl -n rook-nfs-system logs -l app=rook-nfs-operator
```

YugabyteDB operator Quikstart

YugaByte DB is a high-performance distributed SQL database (more information here). Rook provides an operator that can create and manage YugabyteDB clusters.

Prerequisites

Follow these instructions to make your kubernetes cluster ready for `Rook`.

TL;DR

You can create a simple YugabyteDB cluster with below commands. For more detailed instructions, please skip to the Deploy Rook YugabyteDB Operator section.

```
1. cd cluster/examples/kubernetes/yugabytedb
2. kubectl create -f operator.yaml
3. kubectl create -f cluster.yaml
```

Use below commands to observe the created cluster.

```
1. kubectl -n rook-yugabytedb-system get pods
```

Deploy Rook YugabyteDB Operator

To begin with, deploy the Rook YugabyteDB operator, which can create/manage the YugabyteDB cluster. Use following commands to do the same.

```
1. cd cluster/examples/kubernetes/yugabytedb
2. kubectl create -f operator.yaml
```

Observe the rook operator using below command.

```
1. kubectl -n rook-yugabytedb-system get pods
```

Create a simple YugabyteDB cluster

After the Rook YugabyteDB operator is up and running, you can create an object of the custom resource type `ybclusters.yugabytedb.rook.io`. A sample resource specs are present in `cluster.yaml`. You can also browse/modify the contents of `cluster.yaml` according to the configuration options available. Refer YugabyteDB CRD documentation for details on available configuration options.

To create a YugabyteDB cluster, run

```
kubectl create -f cluster.yaml
```

Verify the created custom resource object using

```
kubectl -n rook-yugabytedb get ybclusters.yugabytedb.rook.io
```

Check if the required replicas of Master & TServer are running, run the following command. Tally the Master & TServer pod count against the corresponding replica count you have in `cluster.yaml` . With no change to the replica count, you should see 3 pods each for Master & TServer.

```
kubectl -n rook-yugabytedb get pods
```

Troubleshooting

Skip this section, if the cluster is up & running. Continue to the Access the Database section to access `ysql` api.

If the cluster does not come up, first run following command to take a look at operator logs.

```
kubectl -n rook-yugabytedb-system logs -l app=rook-yugabytedb-operator
```

If everything is OK in the operator logs, check the YugabyteDB Master & TServer logs next.

```
kubectl -n rook-yugabytedb logs -l app=yb-master-hello-ybdb-cluster
kubectl -n rook-yugabytedb logs -l app=yb-tserver-hello-ybdb-cluster
```

Access the Database

After all the pods in YugabyteDB cluster are running, you can access the YugabyteDB's postgres compliant `ysql` api. Run following command to access it.

```
kubectl -n rook-yugabytedb exec -it yb-tserver-hello-ybdb-cluster-0 /home/yugabyte/bin/ysqlsh -- -h yb-
tserver-hello-ybdb-cluster-0  --echo-queries
```

Refer the YugabyteDB documentation for more details on the `ysql` api.

You can also access the YugabyteDB dashboard using port-forwarding.

```
kubectl port-forward -n rook-yugabytedb svc/yb-master-ui-hello-ybdb-cluster 7000:7000
```

> **NOTE**: You should now be able to navigate to `127.0.0.1:7000` to visualize your cluster.

Cleanup

Run the commands below to clean up all resources created above.

> **NOTE**: This will destroy your database and delete all of its data.

```
1. kubectl delete -f cluster.yaml
2. kubectl delete -f operator.yaml
```

Manually delete any Persistent Volumes that were created for this YugabyteDB cluster.

Prerequisites

Rook can be installed on any existing Kubernetes cluster as long as it meets the minimum version and Rook is granted the required privileges (see below for more information). If you don’t have a Kubernetes cluster, you can quickly set one up using Minikube, Kubeadm or CoreOS/Vagrant.

Minimum Version

Kubernetes v1.11 or higher is supported by Rook.

Ceph Prerequisites

See also **Ceph Prerequisites**.

Pod Security Policies

Rook requires privileges to manage the storage in your cluster. If you have Pod Security Policies enabled please review this section. By default, Kubernetes clusters do not have PSPs enabled so you may be able to skip this section.

If you are configuring Ceph on OpenShift, the Ceph walkthrough will configure the PSPs as well when you start the operator with operator-openshift.yaml.

Cluster Role

> **NOTE**: Cluster role configuration is only needed when you are not already `cluster-admin` in your Kubernetes cluster!

Creating the Rook operator requires privileges for setting up RBAC. To launch the operator you need to have created your user certificate that is bound to ClusterRole `cluster-admin` .

One simple way to achieve it is to assign your certificate with the `system:masters` group:

```
1. -subj "/CN=admin/O=system:masters"
```

`system:masters` is a special group that is bound to `cluster-admin` ClusterRole, but it can’t be easily revoked so be careful with taking that route in a production setting. Binding individual certificate to ClusterRole `cluster-admin` is revocable by deleting the ClusterRoleBinding.

RBAC for PodSecurityPolicies

If you have activated the PodSecurityPolicy Admission Controller and thus are using PodSecurityPolicies, you will require additional `(Cluster)RoleBindings` for the different `ServiceAccounts` Rook uses to start the Rook Storage Pods.

Security policies will differ for different backends. See Ceph's Pod Security Policies set up in common.yaml for an example of how this is done in practice.

PodSecurityPolicy

You need at least one `PodSecurityPolicy` that allows privileged `Pod` execution. Here is an example which should be more permissive than is needed for any backend:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: privileged
spec:
  fsGroup:
    rule: RunAsAny
  privileged: true
  runAsUser:
    rule: RunAsAny
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  volumes:
    - '*'
  allowedCapabilities:
    - '*'
  hostPID: true
  # hostNetwork is required for using host networking
  hostNetwork: false
```

Hint: Allowing `hostNetwork` usage is required when using `hostNetwork: true` in a Cluster `CustomResourceDefinition`! You are then also required to allow the usage of `hostPorts` in the `PodSecurityPolicy`. The given port range will allow all ports:

```
   hostPorts:
     # Ceph msgr2 port
     - min: 1
       max: 65535
```

Authenticated docker registries

If you want to use an image from authenticated docker registry (e.g. for image

cache/mirror), you'll need to add an `imagePullSecret` to all relevant service accounts. This way all pods created by the operator (for service account: `rook-ceph-system`) or all new pods in the namespace (for service account: `default`) will have the `imagePullSecret` added to their spec.

The whole process is described in the official kubernetes documentation.

Example setup for a ceph cluster

To get you started, here's a quick rundown for the ceph example from the quickstart guide.

First, we'll create the secret for our registry as described here:

```
# for namespace rook-ceph
kubectl -n rook-ceph create secret docker-registry my-registry-secret --docker-server=DOCKER_REGISTRY_SERVER --docker-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL

# and for namespace rook-ceph (cluster)
kubectl -n rook-ceph create secret docker-registry my-registry-secret --docker-server=DOCKER_REGISTRY_SERVER --docker-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL
```

Next we'll add the following snippet to all relevant service accounts as described here:

```
imagePullSecrets:
- name: my-registry-secret
```

The service accounts are:

- `rook-ceph-system` (namespace: `rook-ceph`): Will affect all pods created by the rook operator in the `rook-ceph` namespace.
- `default` (namespace: `rook-ceph`): Will affect most pods in the `rook-ceph` namespace.
- `rook-ceph-mgr` (namespace: `rook-ceph`): Will affect the MGR pods in the `rook-ceph` namespace.
- `rook-ceph-osd` (namespace: `rook-ceph`): Will affect the OSD pods in the `rook-ceph` namespace.

You can do it either via e.g. `kubectl -n <namespace> edit serviceaccount default` or by modifying the `operator.yaml` and `cluster.yaml` before deploying them.

Since it's the same procedure for all service accounts, here is just one example:

```
kubectl -n rook-ceph edit serviceaccount default
```

```
apiVersion: v1
kind: ServiceAccount
metadata:
```

```
  name: default
  namespace: rook-ceph
secrets:
- name: default-token-12345
imagePullSecrets:               # here are the new
- name: my-registry-secret      # parts
```

After doing this for all service accounts all pods should be able to pull the image from your registry.

Bootstrapping Kubernetes

Rook will run wherever Kubernetes is running. Here are some simple environments to help you get started with Rook.

Minikube

To install `minikube`, refer to this page. Once you have `minikube` installed, start a cluster by doing the following:

```
$ minikube start
Starting local Kubernetes cluster...
Starting VM...
SSH-ing files into VM...
Setting up certs...
Starting cluster components...
Connecting to cluster...
Setting up kubeconfig...
Kubectl is now configured to use the cluster.
```

After these steps, your minikube cluster is ready to install Rook on.

Kubeadm

You can easily spin up Rook on top of a `kubeadm` cluster. You can find the instructions on how to install kubeadm in the Install `kubeadm` page.

By using `kubeadm`, you can use Rook in just a few minutes!

Ceph FlexVolume Configuration

FlexVolume is not enabled by default since Rook v1.1. This documentation applies only if you have enabled FlexVolume.

If enabled, Rook uses FlexVolume to integrate with Kubernetes for performing storage operations. In some operating systems where Kubernetes is deployed, the default Flexvolume plugin directory (the directory where FlexVolume drivers are installed) is **read-only**.

Some Kubernetes deployments require you to configure kubelet with a FlexVolume plugin directory that is accessible and read/write (`rw`). These steps need to be carried out on **all nodes** in your cluster. Rook needs to be told where this directory is in order for the volume plugin to work.

Platform-specific instructions for the following Kubernetes deployment platforms are linked below

- Default FlexVolume path
- Atomic
- Azure AKS
- ContainerLinux
- Google Kubernetes Engine (GKE)
- Kubespray
- OpenShift
- OpenStack Magnum
- Rancher
- Tectonic
- Custom containerized kubelet
- Configuring the FlexVolume path

Default FlexVolume path

If you are not using a platform that is listed above and the path `/usr/libexec/kubernetes/kubelet-plugins/volume/exec/` is read/write, you don't need to configure anything.

That is because `/usr/libexec/kubernetes/kubelet-plugins/volume/exec/` is the kubelet default FlexVolume path and Rook assumes the default FlexVolume path if not set differently.

If running `mkdir -p /usr/libexec/kubernetes/kubelet-plugins/volume/exec/` should give you an error about read-only filesystem, you need to use another read/write FlexVolume path and configure it on the Rook operator and kubelet.

These are the other commonly used paths:

- `/var/lib/kubelet/volumeplugins`
- `/var/lib/kubelet/volume-plugins`

Continue with configuring the FlexVolume path to configure Rook to use the FlexVolume path.

Atomic

See the OpenShift section, unless running with OpenStack Magnum, then see OpenStack Magnum section.

Azure AKS

AKS uses a non-standard FlexVolume plugin directory: `/etc/kubernetes/volumeplugins` The kubelet on AKS is already configured to use that directory.

Continue with configuring the FlexVolume path to configure Rook to use the FlexVolume path.

ContainerLinux

Use the Most common read/write FlexVolume path for the next steps.

The kubelet's systemD unit file can be located at: `/etc/systemd/system/kubelet.service` .

Continue with configuring the FlexVolume path to configure Rook to use the FlexVolume path.

Google Kubernetes Engine (GKE)

Google's Kubernetes Engine uses a non-standard FlexVolume plugin directory: `/home/kubernetes/flexvolume` The kubelet on GKE is already configured to use that directory.

Continue with configuring the FlexVolume path to configure Rook to use the FlexVolume path.

Kubespray

Prior to v2.11.0

Kubespray uses the kubelet_flexvolumes_plugins_dir variable to define where it sets the plugin directory.

Kubespray prior to the v2.11.0 release used a non-standard FlexVolume plugin directory: `/var/lib/kubelet/volume-plugins` . The Kubespray configured kubelet is already

configured to use that directory.

If you are using kubespray v2.10.x or older, continue with configuring the FlexVolume path to configure Rook to use the FlexVolume path.

As of v2.11.0 and newer

Kubespray v2.11.0 included https://github.com/kubernetes-sigs/kubespray/pull/4752 which sets the same plugin directory assumed by rook by default: `/usr/libexec/kubernetes/kubelet-plugins/volume/exec` .

No special configuration of the directory is needed in Rook unless:

- Kubespray is deployed onto a platform where the default path is not writable, or
- you have explicitly defined a custom path in the kubelet_flexvolumes_plugins_dir variable

If you have not defined one, and the default path is not writable, the alternate configuration is `/var/lib/kubelet/volumeplugins`

If needed, continue with configuring the FlexVolume path to configure Rook to use the FlexVolume path.

OpenShift

To find out which FlexVolume directory path you need to set on the Rook operator, please look at the OpenShift docs of the version you are using, latest OpenShift Flexvolume docs (they also contain the FlexVolume path for Atomic).

Continue with configuring the FlexVolume path to configure Rook to use the FlexVolume path.

OpenStack Magnum

OpenStack Magnum is using Atomic, which uses a non-standard FlexVolume plugin directory at: `/var/lib/kubelet/volumeplugins` The kubelet in OpenStack Magnum is already configured to use that directory. You will need to use this value when configuring the Rook operator

Rancher

Rancher provides an easy way to configure kubelet. The FlexVolume flag will be shown later on in the configuring the FlexVolume path. It can be provided to the kubelet configuration template at deployment time or by using the `up to date` feature if Kubernetes is already deployed.

Rancher deploys kubelet as a docker container, you need to mount the host's flexvolume path into the kubelet image as a volume, this can be done in the `extra_binds` section of the kubelet cluster config.

Configure the Rancher deployed kubelet by updating the `cluster.yml` file kubelet section:

```
services:
  kubelet:
    extra_args:
      volume-plugin-dir: /usr/libexec/kubernetes/kubelet-plugins/volume/exec
    extra_binds:
      - /usr/libexec/kubernetes/kubelet-plugins/volume/exec:/usr/libexec/kubernetes/kubelet-
plugins/volume/exec
```

If you're using rke, run `rke up`, this will update and restart your kubernetes cluster system components, in this case the kubelet docker instance(s) will get restarted with the new volume bind and volume plugin dir flag.

The default FlexVolume path for Rancher is `/usr/libexec/kubernetes/kubelet-plugins/volume/exec` which is also the default FlexVolume path for the Rook operator.

If the default path as above is used no further configuration is required, otherwise if a different path is used the Rook operator will need to be reconfigured, to do this continue with configuring the FlexVolume path to configure Rook to use the FlexVolume path.

Tectonic

Follow these instructions to configure the Flexvolume plugin for Rook on Tectonic during ContainerLinux node ignition file provisioning. If you want to use Rook with an already provisioned Tectonic cluster, please refer to the ContainerLinux section.

Continue with configuring the FlexVolume path to configure Rook to use the FlexVolume path.

Custom containerized kubelet

Use the most common read/write FlexVolume path for the next steps.

If your kubelet is running as a (Docker, rkt, etc) container you need to make sure that this directory from the host is reachable by the kubelet inside the container.

Continue with configuring the FlexVolume path to configure Rook to use the FlexVolume path.

Configuring the FlexVolume path

If the environment specific section doesn't mention a FlexVolume path in this doc or external docs, please refer to the most common read/write FlexVolume path section, before continuing to configuring the FlexVolume path.

Configuring the Rook operator

You must provide the above found FlexVolume path when deploying the rook-operator by setting the environment variable `FLEXVOLUME_DIR_PATH` .

Example:

```
spec:
  template:
    spec:
      containers:
[...]
       - name: rook-ceph-operator
         env:
[...]
         - name: FLEXVOLUME_DIR_PATH
           value: "/var/lib/kubelet/volumeplugins"
[...]
```

(In the `operator.yaml` manifest replace `<PathToFlexVolumes>` with the path or if you use helm set the `agent.flexVolumeDirPath` to the FlexVolume path)

Configuring the Kubernetes kubelet

You need to add the flexvolume flag with the path to all nodes's kubelet in the Kubernetes cluster:

```
--volume-plugin-dir=PATH_TO_FLEXVOLUME
```

(Where the `PATH_TO_FLEXVOLUME` is the above found FlexVolume path)

The location where you can set the kubelet FlexVolume path (flag) depends on your platform. Please refer to your platform documentation for that and/or the platform specific FlexVolume path for information about that.

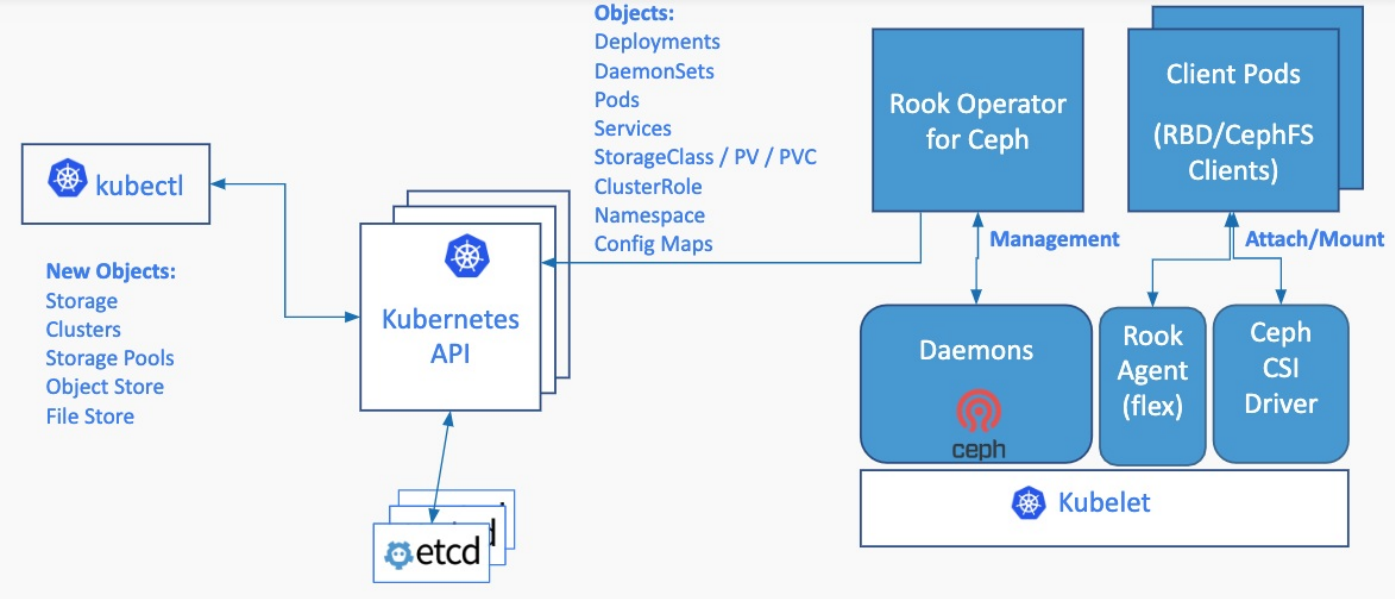
After adding the flag to kubelet, kubelet must be restarted for it to pick up the new flag.

Ceph Storage

Ceph is a highly scalable distributed storage solution for **block storage**, **object storage**, and **shared filesystems** with years of production deployments.

Design

Rook enables Ceph storage systems to run on Kubernetes using Kubernetes primitives. The following image illustrates how Ceph Rook integrates with Kubernetes:

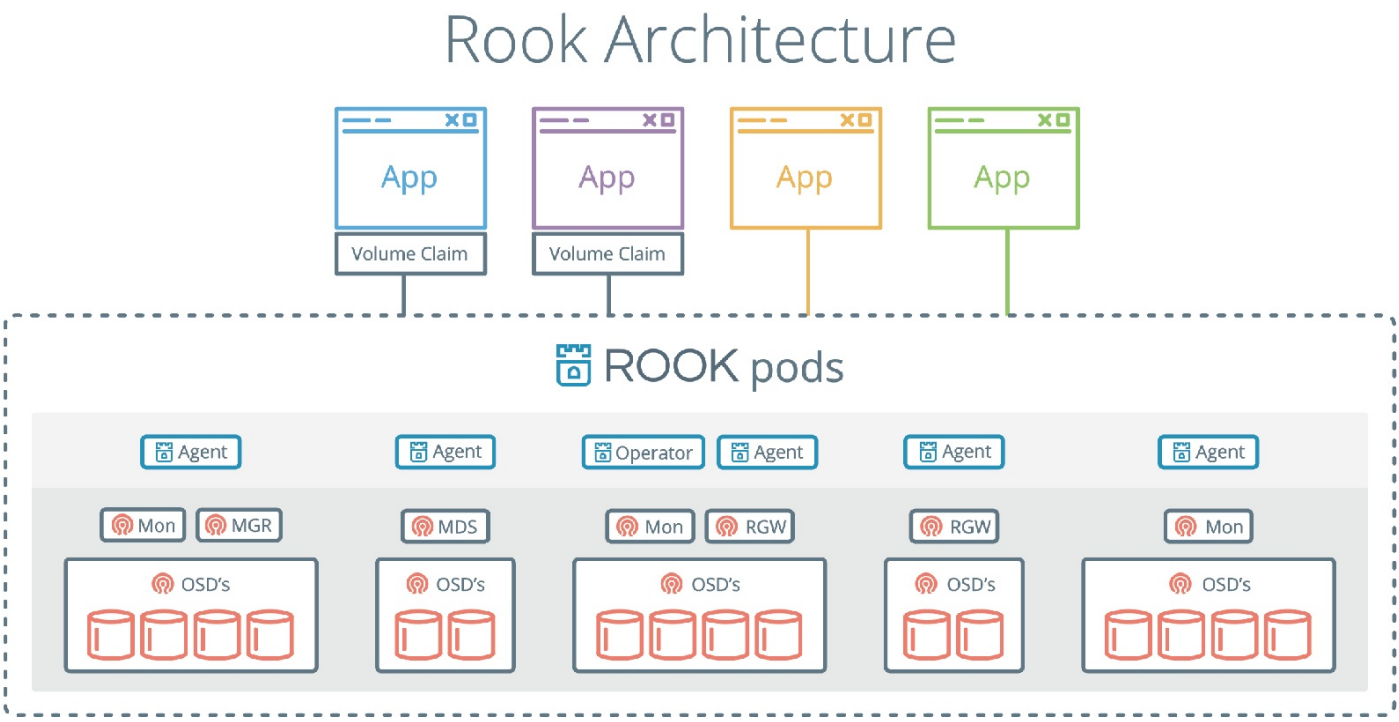


With Ceph running in the Kubernetes cluster, Kubernetes applications can mount block devices and filesystems managed by Rook, or can use the S3/Swift API for object storage. The Rook operator automates configuration of storage components and monitors the cluster to ensure the storage remains available and healthy.

The Rook operator is a simple container that has all that is needed to bootstrap and monitor the storage cluster. The operator will start and monitor Ceph monitor pods, the Ceph OSD daemons to provide RADOS storage, as well as start and manage other Ceph daemons. The operator manages CRDs for pools, object stores (S3/Swift), and filesystems by initializing the pods and other artifacts necessary to run the services.

The operator will monitor the storage daemons to ensure the cluster is healthy. Ceph mons will be started or failed over when necessary, and other adjustments are made as the cluster grows or shrinks. The operator will also watch for desired state changes requested by the api service and apply the changes.

The Rook operator also initializes the agents that are needed for consuming the storage. Rook automatically configures the Ceph-CSI driver to mount the storage to your pods. Rook’s flex driver is still also configured automatically, though will soon be deprecated in favor of the CSI driver.



The `rook/ceph` image includes all necessary tools to manage the cluster – there is no change to the data path. Rook does not attempt to maintain full fidelity with Ceph. Many of the Ceph concepts like placement groups and crush maps are hidden so you don't have to worry about them. Instead Rook creates a much simplified user experience for admins that is in terms of physical resources, pools, volumes, filesystems, and buckets. At the same time, advanced configuration can be applied when needed with the Ceph tools.

Rook is implemented in golang. Ceph is implemented in C++ where the data path is highly optimized. We believe this combination offers the best of both worlds.

Ceph Prerequisites

To make sure you have a Kubernetes cluster that is ready for `Rook`, review the general Rook Prerequisites.

In order to configure the Ceph storage cluster, at least one of these local storage options are required:

- Raw devices (no partitions or formatted filesystems)
- Raw partitions (no formatted filesystem)
- PVs available from a storage class in `block` mode

LVM package

Ceph OSDs have a dependency on LVM in the following scenarios:

- OSDs are created on raw devices or partitions
- If encryption is enabled (`encryptedDevice: true` in the cluster CR)
- A `metadata` device is specified

LVM is not required for OSDs in these scenarios:

- Creating OSDs on PVCs using the `storageClassDeviceSets`

If LVM is required for your scenario, LVM needs to be available on the hosts where OSDs will be running. Some Linux distributions do not ship with the `lvm2` package. This package is required on all storage nodes in your k8s cluster to run Ceph OSDs. Without this package even though Rook will be able to successfully create the Ceph OSDs, when a node is rebooted the OSD pods running on the restarted node will **fail to start**. Please install LVM using your Linux distribution's package manager. For example:

CentOS:

```
1. sudo yum install -y lvm2
```

Ubuntu:

```
1. sudo apt-get install -y lvm2
```

RancherOS:

- Since version 1.5.0 LVM is supported
- Logical volumes will not be activated during the boot process. You need to add an runcmd command for that.

```
runcmd:
- [ vgchange, -ay ]
```

Ceph Flexvolume Configuration

NOTE This configuration is only needed when using the FlexVolume driver (required for Kubernetes 1.12 or earlier). The Ceph-CSI RBD driver or the Ceph-CSI CephFS driver are recommended for Kubernetes 1.13 and newer, making FlexVolume configuration redundant.

If you want to configure volumes with the Flex driver instead of CSI, the Rook agent requires setup as a Flex volume plugin to manage the storage attachments in your cluster. See the Flex Volume Configuration topic to configure your Kubernetes deployment to load the Rook volume plugin.

Extra agent mounts

On certain distributions it may be necessary to mount additional directories into the agent container. That is what the environment variable `AGENT_MOUNTS` is for. Also see the documentation in helm-operator on the parameter `agent.mounts`. The format of the variable content should be `mountname1=/host/path1:/container/path1,mountname2=/host/path2:/container/path2`.

Kernel

RBD

Ceph requires a Linux kernel built with the RBD module. Many Linux distributions have this module, but not all distributions. For example, the GKE Container-Optimised OS (COS) does not have RBD.

You can test your Kubernetes nodes by running `modprobe rbd`. If it says 'not found', you may have to rebuild your kernel or choose a different Linux distribution.

CephFS

If you will be creating volumes from a Ceph shared file system (CephFS), the recommended minimum kernel version is **4.17**. If you have a kernel version less than 4.17, the requested PVC sizes will not be enforced. Storage quotas will only be enforced on newer kernels.

Kernel modules directory configuration

Normally, on Linux, kernel modules can be found in `/lib/modules`. However, there are some distributions that put them elsewhere. In that case the environment variable

`LIB_MODULES_DIR_PATH` can be used to override the default. Also see the documentation in helm-operator on the parameter `agent.libModulesDirPath` . One notable distribution where this setting is useful would be NixOS.

Admission Controller

An admission controller intercepts requests to the Kubernetes API server prior to persistence of the object, but after the request is authenticated and authorized.

Enabling the Rook admission controller is recommended to provide an additional level of validation that Rook is configured correctly with the custom resource (CR) settings.

Quick Start

To deploy the Rook admission controllers we have a helper script that will automate the configuration.

This script will help us achieve the following tasks

1. Creates self-signed certificate.
2. Creates a Certificate Signing Request(CSR) for the certificate and gets it approved from the Kubernetes cluster.
3. Stores these certificates as a Kubernetes Secret.
4. Creates a Service Account, ClusterRole and ClusterRoleBindings for running the webhook server with minimal priviledges.
5. Creates ValidatingWebhookConfig and fills the CA bundle with the appropriate value from the cluster.

Run the following commands:

```
1. kubectl create -f examples/kubernetes/ceph/common.yaml
2. cluster/examples/kubernetes/ceph/config-admission-controller.sh
```

Now that the Secrets have been deployed, we can deploy the operator:

```
1. kubectl create -f operator.yaml
```

At this point the operator will start the admission controller Deployment automatically and the Webhook will start intercepting requests for Rook resources.

Certificate Management

```
    The script file creates a self-signed Kubernetes approved certificate and deploys it as a secret onto the
    cluster. It is mandatory that the Secret is named "rook-ceph-admission-controller" because Rook will look for
1. the secret with such name before starting the admission controller servers.
```

In the case of deploying from scratch, the script needs to be executed once without

any modification, and certificates will be automatically created and deployed as a Secret.

The above approach of using self-signed certificates is discouraged as it would be the job of the owner to maintain the certificates. The recommended approach would be to use a proper certificate manager and get signed certificates from a known certificate authority. Once these are available, create the secrets using the following command:

```
kubectl create secret generic rook-ceph-admission-controller \
        --from-file=key.pem=${PRIVATE_KEY_NAME}.pem \
        --from-file=cert.pem=${PUBLIC_KEY_NAME}.pem
```

Once the Secrets are in the cluster, we can modify the parameter `INSTALL_SELF_SIGNED_CERT` to `false` and execute these scripts to deploy the components. This modification is required only when Secrets are created but the components (ValidatingWebhookConfig, RBAC) are yet to be deployed.

Ceph Examples

Configuration for Rook and Ceph can be configured in multiple ways to provide block devices, shared filesystem volumes or object storage in a kubernetes namespace. We have provided several examples to simplify storage setup, but remember there are many tunables and you will need to decide what settings work for your use case and environment.

See the **example yaml files** folder for all the rook/ceph setup example spec files.

Common Resources

The first step to deploy Rook is to create the common resources. The configuration for these resources will be the same for most deployments. The common.yaml sets these resources up.

```
1. kubectl create -f common.yaml
```

The examples all assume the operator and all Ceph daemons will be started in the same namespace. If you want to deploy the operator in a separate namespace, see the comments throughout `common.yaml` .

Operator

After the common resources are created, the next step is to create the Operator deployment. Several spec file examples are provided in this directory:

- `operator.yaml` : The most common settings for production deployments
  - `kubectl create -f operator.yaml`
- `operator-openshift.yaml` : Includes all of the operator settings for running a basic Rook cluster in an OpenShift environment. You will also want to review the OpenShift Prerequisites to confirm the settings.
  - `oc create -f operator-openshift.yaml`

Settings for the operator are configured through environment variables on the operator deployment. The individual settings are documented in operator.yaml.

Cluster CRD

Now that your operator is running, let’s create your Ceph storage cluster. This CR contains the most critical settings that will influence how the operator configures the storage. It is important to understand the various ways to configure the cluster. These examples represent a very small set of the different ways to configure the

storage.

- `cluster.yaml`: This file contains common settings for a production storage cluster. Requires at least three nodes.
- `cluster-test.yaml`: Settings for a test cluster where redundancy is not configured. Requires only a single node.
- `cluster-on-pvc.yaml`: This file contains common settings for backing the Ceph Mons and OSDs by PVs. Useful when running in cloud environments or where local PVs have been created for Ceph to consume.
- `cluster-with-drive-groups.yaml`: This file contains example configurations for creating advanced OSD layouts on nodes using Ceph Drive Groups. See docs for more
- `cluster-external`: Connect to an external Ceph cluster with minimal access to monitor the health of the cluster and connect to the storage.
- `cluster-external-management`: Connect to an external Ceph cluster with the admin key of the external cluster to enable remote creation of pools and configure services such as an Object Store or a Shared Filesystem.

See the Cluster CRD topic for more details and more examples for the settings.

Setting up consumable storage

Now we are ready to setup block, shared filesystem or object storage in the Rook Ceph cluster. These kinds of storage are respectively referred to as CephBlockPool, CephFilesystem and CephObjectStore in the spec files.

Block Devices

Ceph can provide raw block device volumes to pods. Each example below sets up a storage class which can then be used to provision a block device in kubernetes pods. The storage class is defined with a pool which defines the level of data redundancy in Ceph:

- `storageclass.yaml`: This example illustrates replication of 3 for production scenarios and requires at least three nodes. Your data is replicated on three different kubernetes worker nodes and intermittent or long-lasting single node failures will not result in data unavailability or loss.
- `storageclass-ec.yaml`: Configures erasure coding for data durability rather than replication. Ceph's erasure coding is more efficient than replication so you can get high reliability without the 3x replication cost of the preceding example (but at the cost of higher computational encoding and decoding costs on the worker nodes). Erasure coding requires at least three nodes. See the Erasure coding documentation for more details.
- `storageclass-test.yaml`: Replication of 1 for test scenarios and it requires only a single node. Do not use this for applications that store valuable data or have high-availability storage requirements, since a single node failure can result in data loss.

The storage classes are found in different sub-directories depending on the driver:

- `csi/rbd`: The CSI driver for block devices. This is the preferred driver going forward.
- `flex`: The flex driver will be deprecated in a future release to be determined.

See the Ceph Pool CRD topic for more details on the settings.

Shared Filesystem

Ceph filesystem (CephFS) allows the user to 'mount' a shared posix-compliant folder into one or more hosts (pods in the container world). This storage is similar to NFS shared storage or CIFS shared folders, as explained here.

File storage contains multiple pools that can be configured for different scenarios:

- `filesystem.yaml`: Replication of 3 for production scenarios. Requires at least three nodes.
- `filesystem-ec.yaml`: Erasure coding for production scenarios. Requires at least three nodes.
- `filesystem-test.yaml`: Replication of 1 for test scenarios. Requires only a single node.

Dynamic provisioning is possible with the CSI driver. The storage class for shared filesystems is found in the `csi/cephfs` directory.

See the Shared Filesystem CRD topic for more details on the settings.

Object Storage

Ceph supports storing blobs of data called objects that support HTTP(s)-type get/put/post and delete semantics. This storage is similar to AWS S3 storage, for example.

Object storage contains multiple pools that can be configured for different scenarios:

- `object.yaml`: Replication of 3 for production scenarios. Requires at least three nodes.
- `object-openshift.yaml`: Replication of 3 with rgw in a port range valid for OpenShift. Requires at least three nodes.
- `object-ec.yaml`: Erasure coding rather than replication for production scenarios. Requires at least three nodes.
- `object-test.yaml`: Replication of 1 for test scenarios. Requires only a single node.

See the Object Store CRD topic for more details on the settings.

Object Storage User

- `object-user.yaml`: Creates a simple object storage user and generates credentials for

the S3 API

Object Storage Buckets

The Ceph operator also runs an object store bucket provisioner which can grant access to existing buckets or dynamically provision new buckets.

- object-bucket-claim-retain.yaml Creates a request for a new bucket by referencing a StorageClass which saves the bucket when the initiating OBC is deleted.
- object-bucket-claim-delete.yaml Creates a request for a new bucket by referencing a StorageClass which deletes the bucket when the initiating OBC is deleted.
- storageclass-bucket-retain.yaml Creates a new StorageClass which defines the Ceph Object Store, a region, and retains the bucket after the initiating OBC is deleted.
- storageclass-bucket-delete.yaml Creates a new StorageClass which defines the Ceph Object Store, a region, and deletes the bucket after the initiating OBC is deleted.

OpenShift

OpenShift adds a number of security and other enhancements to Kubernetes. In particular, security context constraints allow the cluster admin to define exactly which permissions are allowed to pods running in the cluster. You will need to define those permissions that allow the Rook pods to run.

The settings for Rook in OpenShift are described below, and are also included in the example yaml files:

- `operator-openshift.yaml`: Creates the security context constraints and starts the operator deployment
- `object-openshift.yaml`: Creates an object store with rgw listening on a valid port number for OpenShift

TL;DR

To create an OpenShift cluster, the commands basically include:

```
oc create -f common.yaml
oc create -f operator-openshift.yaml
oc create -f cluster.yaml
```

Rook Privileges

To orchestrate the storage platform, Rook requires the following access in the cluster:

- Create `hostPath` volumes, for persistence by the Ceph mon and osd pods
- Run pods in `privileged` mode, for access to `/dev` and `hostPath` volumes
- Host networking for the Rook agent and clusters that require host networking
- Ceph OSDs require host PIDs for communication on the same node

Security Context Constraints

Before starting the Rook operator or cluster, create the security context constraints needed by the Rook pods. The following yaml is found in `operator-openshift.yaml` under `/cluster/examples/kubernetes/ceph`.

> **NOTE**: Older versions of OpenShift may require `apiVersion: v1`.

```
kind: SecurityContextConstraints
apiVersion: security.openshift.io/v1
metadata:
```

```
  name: rook-ceph
allowPrivilegedContainer: true
allowHostNetwork: true
allowHostDirVolumePlugin: true
priority:
allowedCapabilities: []
allowHostPorts: false
allowHostPID: true
allowHostIPC: false
readOnlyRootFilesystem: false
requiredDropCapabilities: []
defaultAddCapabilities: []
runAsUser:
  type: RunAsAny
seLinuxContext:
  type: MustRunAs
fsGroup:
  type: MustRunAs
supplementalGroups:
  type: RunAsAny
allowedFlexVolumes:
  - driver: "ceph.rook.io/rook"
  - driver: "ceph.rook.io/rook-ceph"
volumes:
  - configMap
  - downwardAPI
  - emptyDir
  - flexVolume
  - hostPath
  - persistentVolumeClaim
  - projected
  - secret
users:
  # A user needs to be added for each rook service account.
  # This assumes running in the default sample "rook-ceph" namespace.
  # If other namespaces or service accounts are configured, they need to be updated here.
  - system:serviceaccount:rook-ceph:rook-ceph-system
  - system:serviceaccount:rook-ceph:default
  - system:serviceaccount:rook-ceph:rook-ceph-mgr
  - system:serviceaccount:rook-ceph:rook-ceph-osd
```

Important to note is that if you plan on running Rook in namespaces other than the default `rook-ceph`, the example scc will need to be modified to accommodate for your namespaces where the Rook pods are running.

To create the scc you will need a privileged account:

```
oc login -u system:admin
```

We will create the security context constraints with the operator in the next section.

Rook Settings

There are some Rook settings that also need to be adjusted to work in OpenShift.

Operator Settings

There is an environment variable that needs to be set in the operator spec that will allow Rook to run in OpenShift clusters.

- `ROOK_HOSTPATH_REQUIRES_PRIVILEGED`: Must be set to `true`. Writing to the hostPath is required for the Ceph mon and osd pods. Given the restricted permissions in OpenShift with SELinux, the pod must be running privileged in order to write to the hostPath volume.

```
- name: ROOK_HOSTPATH_REQUIRES_PRIVILEGED
  value: "true"
```

Now create the security context constraints and the operator:

```
oc create -f operator-openshift.yaml
```

Cluster Settings

The cluster settings in `cluster.yaml` are largely isolated from the differences in OpenShift. There is perhaps just one to take note of:

- `dataDirHostPath`: Ensure that it points to a valid, writable path on the host systems.

Object Store Settings

In OpenShift, ports less than 1024 cannot be bound. In the object store CRD, ensure the port is modified to meet this requirement.

```
gateway:
  port: 8080
```

You can expose a different port such as `80` by creating a service.

A sample object store can be created with these settings:

```
oc create -f object-openshift.yaml
```

Block Storage

Block storage allows a single pod to mount storage. This guide shows how to create a simple, multi-tier web application on Kubernetes using persistent volumes enabled by Rook.

Prerequisites

This guide assumes a Rook cluster as explained in the Quickstart.

Provision Storage

Before Rook can provision storage, a `StorageClass` and `CephBlockPool` need to be created. This will allow Kubernetes to interoperate with Rook when provisioning persistent volumes.

> **NOTE**: This sample requires *at least 1 OSD per node*, with each OSD located on *3 different nodes*.

Each OSD must be located on a different node, because the `failureDomain` is set to `host` and the `replicated.size` is set to `3`.

> **NOTE**: This example uses the CSI driver, which is the preferred driver going forward for K8s 1.13 and newer. Examples are found in the CSI RBD directory. For an example of a storage class using the flex driver (required for K8s 1.12 or earlier), see the Flex Driver section below, which has examples in the flex directory.

Save this `StorageClass` definition as `storageclass.yaml`:

```
apiVersion: ceph.rook.io/v1
kind: CephBlockPool
metadata:
  name: replicapool
  namespace: rook-ceph
spec:
  failureDomain: host
  replicated:
    size: 3
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: rook-ceph-block
# Change "rook-ceph" provisioner prefix to match the operator namespace if needed
provisioner: rook-ceph.rbd.csi.ceph.com
parameters:
    # clusterID is the namespace where the rook cluster is running
    clusterID: rook-ceph
    # Ceph pool into which the RBD image shall be created
```

```
    pool: replicapool

    # RBD image format. Defaults to "2".
    imageFormat: "2"

    # RBD image features. Available for imageFormat: "2". CSI RBD currently supports only `layering` feature.
    imageFeatures: layering

    # The secrets contain Ceph admin credentials.
    csi.storage.k8s.io/provisioner-secret-name: rook-csi-rbd-provisioner
    csi.storage.k8s.io/provisioner-secret-namespace: rook-ceph
    csi.storage.k8s.io/controller-expand-secret-name: rook-csi-rbd-provisioner
    csi.storage.k8s.io/controller-expand-secret-namespace: rook-ceph
    csi.storage.k8s.io/node-stage-secret-name: rook-csi-rbd-node
    csi.storage.k8s.io/node-stage-secret-namespace: rook-ceph

    # Specify the filesystem type of the volume. If not specified, csi-provisioner
    # will set default as `ext4`. Note that `xfs` is not recommended due to potential deadlock
    # in hyperconverged settings where the volume is mounted on the same node as the osds.
    csi.storage.k8s.io/fstype: ext4

# Delete the rbd volume when a PVC is deleted
reclaimPolicy: Delete
```

If you've deployed the Rook operator in a namespace other than "rook-ceph", change the prefix in the provisioner to match the namespace you used. For example, if the Rook operator is running in the namespace "my-namespace" the provisioner value should be "my-namespace.rbd.csi.ceph.com".

Create the storage class.

```
kubectl create -f cluster/examples/kubernetes/ceph/csi/rbd/storageclass.yaml
```

> **NOTE**: As specified by Kubernetes, when using the `Retain` reclaim policy, any Ceph RBD image that is backed by a `PersistentVolume` will continue to exist even after the `PersistentVolume` has been deleted. These Ceph RBD images will need to be cleaned up manually using `rbd rm`.

Consume the storage: Wordpress sample

We create a sample app to consume the block storage provisioned by Rook with the classic wordpress and mysql apps. Both of these apps will make use of block volumes provisioned by Rook.

Start mysql and wordpress from the `cluster/examples/kubernetes` folder:

```
kubectl create -f mysql.yaml
kubectl create -f wordpress.yaml
```

Both of these apps create a block volume and mount it to their respective pod. You can

see the Kubernetes volume claims by running the following:

```
$ kubectl get pvc
NAME             STATUS    VOLUME                                     CAPACITY   ACCESSMODES   AGE
mysql-pv-claim   Bound     pvc-95402dbc-efc0-11e6-bc9a-0cc47a3459ee   20Gi       RWO           1m
wp-pv-claim      Bound     pvc-39e43169-efc1-11e6-bc9a-0cc47a3459ee   20Gi       RWO           1m
```

Once the wordpress and mysql pods are in the `Running` state, get the cluster IP of the wordpress app and enter it in your browser:

```
$ kubectl get svc wordpress
NAME        CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
wordpress   10.3.0.155   <pending>     80:30841/TCP   2m
```

You should see the wordpress app running.

If you are using Minikube, the Wordpress URL can be retrieved with this one-line command:

```
echo http://$(minikube ip):$(kubectl get service wordpress -o jsonpath='{.spec.ports[0].nodePort}')
```

> **NOTE**: When running in a vagrant environment, there will be no external IP address to reach wordpress with. You will only be able to reach wordpress via the `CLUSTER-IP` from inside the Kubernetes cluster.

Consume the storage: Toolbox

With the pool that was created above, we can also create a block image and mount it directly in a pod. See the Direct Block Tools topic for more details.

Teardown

To clean up all the artifacts created by the block demo:

```
kubectl delete -f wordpress.yaml
kubectl delete -f mysql.yaml
kubectl delete -n rook-ceph cephblockpools.ceph.rook.io replicapool
kubectl delete storageclass rook-ceph-block
```

Flex Driver

To create a volume based on the flex driver instead of the CSI driver, see the following example of a storage class. Make sure the flex driver is enabled over Ceph CSI. For this, you need to set `ROOK_ENABLE_FLEX_DRIVER` to `true` in your operator deployment in the `operator.yaml` file. The pool definition is the same as for the CSI

driver.

```
apiVersion: ceph.rook.io/v1
kind: CephBlockPool
metadata:
  name: replicapool
  namespace: rook-ceph
spec:
  failureDomain: host
  replicated:
    size: 3
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: rook-ceph-block
provisioner: ceph.rook.io/block
parameters:
  blockPool: replicapool
  # The value of "clusterNamespace" MUST be the same as the one in which your rook cluster exist
  clusterNamespace: rook-ceph
  # Specify the filesystem type of the volume. If not specified, it will use `ext4`.
  fstype: ext4
# Optional, default reclaimPolicy is "Delete". Other options are: "Retain", "Recycle" as documented in
https://kubernetes.io/docs/concepts/storage/storage-classes/
reclaimPolicy: Retain
# Optional, if you want to add dynamic resize for PVC. Works for Kubernetes 1.14+
# For now only ext3, ext4, xfs resize support provided, like in Kubernetes itself.
allowVolumeExpansion: true
```

Create the pool and storage class using `kubectl` :

```
kubectl create -f cluster/examples/kubernetes/ceph/flex/storageclass.yaml
```

Continue with the example above for the wordpress application.

Advanced Example: Erasure Coded Block Storage

If you want to use erasure coded pool with RBD, your OSDs must use `bluestore` as their `storeType` . Additionally the nodes that are going to mount the erasure coded RBD block storage must have Linux kernel >= `4.11` .

NOTE: This example requires *at least 3 bluestore OSDs*, with each OSD located on a *different node*.

The OSDs must be located on different nodes, because the `failureDomain` is set to `host` and the `erasureCoded` chunk settings require at least 3 different OSDs (2 `dataChunks` + 1 `codingChunks`).

To be able to use an erasure coded pool you need to create two pools (as seen below in

the definitions): one erasure coded and one replicated.

> **NOTE**: This example requires *at least 3 bluestore OSDs*, with each OSD located on a *different node*.

The OSDs must be located on different nodes, because the `failureDomain` is set to `host` and the `erasureCoded` chunk settings require at least 3 different OSDs (2 `dataChunks` + 1 `codingChunks`).

Erasure Coded CSI Driver

The erasure coded pool must be set as the `dataPool` parameter in `storageclass-ec.yaml` It is used for the data of the RBD images.

Erasure Coded Flex Driver

The erasure coded pool must be set as the `dataBlockPool` parameter in `storageclass-ec.yaml` . It is used for the data of the RBD images.

Object Storage

Object storage exposes an S3 API to the storage cluster for applications to put and get data.

Prerequisites

This guide assumes a Rook cluster as explained in the Quickstart.

Configure an Object Store

Rook has the ability to either deploy an object store in Kubernetes or to connect to an external RGW service. Most commonly, the object store will be configured locally by Rook. Alternatively, if you have an existing Ceph cluster with Rados Gateways, see the external section to consume it from Rook.

Create a Local Object Store

The below sample will create a `CephObjectStore` that starts the RGW service in the cluster with an S3 API.

> **NOTE**: This sample requires *at least 3 bluestore OSDs*, with each OSD located on a *different node*.

The OSDs must be located on different nodes, because the `failureDomain` is set to `host` and the `erasureCoded` chunk settings require at least 3 different OSDs (2 `dataChunks` + 1 `codingChunks`).

See the Object Store CRD, for more detail on the settings available for a `CephObjectStore`.

```
1.  apiVersion: ceph.rook.io/v1
2.  kind: CephObjectStore
3.  metadata:
4.    name: my-store
5.    namespace: rook-ceph
6.  spec:
7.    metadataPool:
8.      failureDomain: host
9.      replicated:
10.       size: 3
11.   dataPool:
12.     failureDomain: host
13.     erasureCoded:
14.       dataChunks: 2
15.       codingChunks: 1
16.   preservePoolsOnDelete: true
```

```
  gateway:
    type: s3
    sslCertificateRef:
    port: 80
    securePort:
    instances: 1
  healthCheck:
    bucket:
      enabled: true
      interval: 60s
```

After the `CephObjectStore` is created, the Rook operator will then create all the pools and other resources necessary to start the service. This may take a minute to complete.

```
# Create the object store
kubectl create -f object.yaml

# To confirm the object store is configured, wait for the rgw pod to start
kubectl -n rook-ceph get pod -l app=rook-ceph-rgw
```

Connect to an External Object Store

Rook can connect to existing RGW gateways to work in conjunction with the external mode of the `CephCluster` CRD. If you have an external `CephCluster` CR, you can instruct Rook to consume external gateways with the following:

```
apiVersion: ceph.rook.io/v1
kind: CephObjectStore
metadata:
  name: external-store
  namespace: rook-ceph
spec:
  gateway:
    port: 8080
    externalRgwEndpoints:
      - ip: 192.168.39.182
  healthCheck:
    bucket:
      enabled: true
      interval: 60s
```

You can use the existing `object-external.yaml` file. When ready the ceph-object-controller will output a message in the Operator log similar to this one:

```
ceph-object-controller: ceph object store gateway service running at 10.100.28.138:8080
```

You can now get and access the store via:

```
kubectl -n rook-ceph get svc -l app=rook-ceph-rgw
NAME                     TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
rook-ceph-rgw-my-store   ClusterIP   10.100.28.138   <none>        8080/TCP   6h59m
```

Any pod from your cluster can now access this endpoint:

```
curl 10.100.28.138:8080
<?xml version="1.0" encoding="UTF-8"?><ListAllMyBucketsResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
<Owner><ID>anonymous</ID><DisplayName></DisplayName></Owner><Buckets></Buckets></ListAllMyBucketsResult>
```

It is also possible to use the internally registered DNS name:

```
curl rook-ceph-rgw-my-store.rook-ceph:8080
<?xml version="1.0" encoding="UTF-8"?><ListAllMyBucketsResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
<Owner><ID>anonymous</ID><DisplayName></DisplayName></Owner><Buckets></Buckets></ListAllMyBucketsResult>
```

The DNS name is created with the following schema `rook-ceph-rgw-$STORE_NAME.$NAMESPACE` .

Create a Bucket

Now that the object store is configured, next we need to create a bucket where a client can read and write objects. A bucket can be created by defining a storage class, similar to the pattern used by block and file storage. First, define the storage class that will allow object clients to create a bucket. The storage class defines the object storage system, the bucket retention policy, and other properties required by the administrator. Save the following as `storageclass-bucket-delete.yaml` (the example is named as such due to the `Delete` reclaim policy).

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: rook-ceph-bucket
provisioner: rook-ceph.ceph.rook.io/bucket
reclaimPolicy: Delete
parameters:
  objectStoreName: my-store
  objectStoreNamespace: rook-ceph
  region: us-east-1
```

```
kubectl create -f storageclass-bucket-delete.yaml
```

Based on this storage class, an object client can now request a bucket by creating an Object Bucket Claim (OBC). When the OBC is created, the Rook-Ceph bucket provisioner will create a new bucket. Notice that the OBC references the storage class that was created above. Save the following as `object-bucket-claim-delete.yaml` (the example is named as such due to the `Delete` reclaim policy):

```
apiVersion: objectbucket.io/v1alpha1
kind: ObjectBucketClaim
metadata:
  name: ceph-bucket
spec:
  generateBucketName: ceph-bkt
  storageClassName: rook-ceph-bucket
```

```
kubectl create -f object-bucket-claim-delete.yaml
```

Now that the claim is created, the operator will create the bucket as well as generate other artifacts to enable access to the bucket. A secret and ConfigMap are created with the same name as the OBC and in the same namespace. The secret contains credentials used by the application pod to access the bucket. The ConfigMap contains bucket endpoint information and is also consumed by the pod. See the Object Bucket Claim Documentation for more details on the `CephObjectBucketClaims` .

Client Connections

The following commands extract key pieces of information from the secret and configmap:”

```
#config-map, secret, OBC will part of default if no specific name space mentioned
export AWS_HOST=$(kubectl -n default get cm ceph-bucket -o yaml | grep BUCKET_HOST | awk '{print $2}')
export AWS_ACCESS_KEY_ID=$(kubectl -n default get secret ceph-bucket -o yaml | grep AWS_ACCESS_KEY_ID | awk '{print $2}' | base64 --decode)
export AWS_SECRET_ACCESS_KEY=$(kubectl -n default get secret ceph-bucket -o yaml | grep AWS_SECRET_ACCESS_KEY | awk '{print $2}' | base64 --decode)
```

Consume the Object Storage

Now that you have the object store configured and a bucket created, you can consume the object storage from an S3 client.

This section will guide you through testing the connection to the `CephObjectStore` and uploading and downloading from it. Run the following commands after you have connected to the Rook toolbox.

Connection Environment Variables

To simplify the s3 client commands, you will want to set the four environment variables for use by your client (ie. inside the toolbox). See above for retrieving the variables for a bucket created by an `ObjectBucketClaim` .

```
export AWS_HOST=<host>
export AWS_ENDPOINT=<endpoint>
export AWS_ACCESS_KEY_ID=<accessKey>
```

```
4. export AWS_SECRET_ACCESS_KEY=<secretKey>
```

- `Host`: The DNS host name where the rgw service is found in the cluster. Assuming you are using the default `rook-ceph` cluster, it will be `rook-ceph-rgw-my-store.rook-ceph`.
- `Endpoint`: The endpoint where the rgw service is listening. Run `kubectl -n rook-ceph get svc rook-ceph-rgw-my-store`, then combine the clusterIP and the port.
- `Access key`: The user's `access_key` as printed above
- `Secret key`: The user's `secret_key` as printed above

The variables for the user generated in this example might be:

```
1. export AWS_HOST=rook-ceph-rgw-my-store.rook-ceph
2. export AWS_ENDPOINT=10.104.35.31:80
3. export AWS_ACCESS_KEY_ID=XEZDB3UJ6X7HVBE7X7MA
4. export AWS_SECRET_ACCESS_KEY=7yGIZON7EhFORz0I40BFniML36D2rl8CQQ5kXU6l
```

The access key and secret key can be retrieved as described in the section above on client connections or below in the section creating a user if you are not creating the buckets with an `ObjectBucketClaim`.

Install s3cmd

To test the `CephObjectStore` we will install the `s3cmd` tool into the toolbox pod.

```
1. yum --assumeyes install s3cmd
```

PUT or GET an object

Upload a file to the newly created bucket

```
1. echo "Hello Rook" > /tmp/rookObj
2. s3cmd put /tmp/rookObj --no-ssl --host=${AWS_HOST} --host-bucket=  s3://rookbucket
```

Download and verify the file from the bucket

```
1. s3cmd get s3://rookbucket/rookObj /tmp/rookObj-download --no-ssl --host=${AWS_HOST} --host-bucket=
2. cat /tmp/rookObj-download
```

Access External to the Cluster

Rook sets up the object storage so pods will have access internal to the cluster. If your applications are running outside the cluster, you will need to setup an external service through a `NodePort`.

First, note the service that exposes RGW internal to the cluster. We will leave this

service intact and create a new service for external access.

```
$ kubectl -n rook-ceph get service rook-ceph-rgw-my-store
NAME                     CLUSTER-IP   EXTERNAL-IP   PORT(S)     AGE
rook-ceph-rgw-my-store   10.3.0.177   <none>        80/TCP      2m
```

Save the external service as `rgw-external.yaml` :

```
apiVersion: v1
kind: Service
metadata:
  name: rook-ceph-rgw-my-store-external
  namespace: rook-ceph
  labels:
    app: rook-ceph-rgw
    rook_cluster: rook-ceph
    rook_object_store: my-store
spec:
  ports:
  - name: rgw
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: rook-ceph-rgw
    rook_cluster: rook-ceph
    rook_object_store: my-store
  sessionAffinity: None
  type: NodePort
```

Now create the external service.

```
kubectl create -f rgw-external.yaml
```

See both rgw services running and notice what port the external service is running on:

```
$ kubectl -n rook-ceph get service rook-ceph-rgw-my-store rook-ceph-rgw-my-store-external
NAME                             TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
rook-ceph-rgw-my-store           ClusterIP   10.104.82.228    <none>        80/TCP         4m
rook-ceph-rgw-my-store-external  NodePort    10.111.113.237   <none>        80:31536/TCP   39s
```

Internally the rgw service is running on port `80` . The external port in this case is `31536` . Now you can access the `CephObjectStore` from anywhere! All you need is the hostname for any machine in the cluster, the external port, and the user credentials.

Create a User

If you need to create an independent set of user credentials to access the S3

endpoint, create a `CephObjectStoreUser`. The user will be used to connect to the RGW service in the cluster using the S3 API. The user will be independent of any object bucket claims that you might have created in the earlier instructions in this document.

See the Object Store User CRD for more detail on the settings available for a `CephObjectStoreUser`.

```
apiVersion: ceph.rook.io/v1
kind: CephObjectStoreUser
metadata:
  name: my-user
  namespace: rook-ceph
spec:
  store: my-store
  displayName: "my display name"
```

When the `CephObjectStoreUser` is created, the Rook operator will then create the RGW user on the specified `CephObjectStore` and store the Access Key and Secret Key in a kubernetes secret in the same namespace as the `CephObjectStoreUser`.

```
# Create the object store user
kubectl create -f object-user.yaml

# To confirm the object store user is configured, describe the secret
kubectl -n rook-ceph describe secret rook-ceph-object-user-my-store-my-user

Name:         rook-ceph-object-user-my-store-my-user
Namespace:    rook-ceph
Labels:           app=rook-ceph-rgw
                  rook_cluster=rook-ceph
                  rook_object_store=my-store
Annotations:    <none>

Type:    kubernetes.io/rook

Data
====
AccessKey:    20 bytes
SecretKey:    40 bytes
```

The AccessKey and SecretKey data fields can be mounted in a pod as an environment variable. More information on consuming kubernetes secrets can be found in the K8s secret documentation

To directly retrieve the secrets:

```
kubectl -n rook-ceph get secret rook-ceph-object-user-my-store-my-user -o yaml | grep AccessKey | awk '{print $2}' | base64 --decode
```

```
kubectl -n rook-ceph get secret rook-ceph-object-user-my-store-my-user -o yaml | grep SecretKey | awk '{print $2}' | base64 --decode
```

Object Multisite

Multisite is a feature of Ceph that allows object stores to replicate its data over multiple Ceph clusters.

Multisite also allows object stores to be independent and isloated from other object stores in a cluster.

For more information on multisite please read the ceph multisite overview for how to run it.

Object Multisite

Multisite is a feature of Ceph that allows object stores to replicate their data over multiple Ceph clusters.

Multisite also allows object stores to be independent and isolated from other object stores in a cluster.

When a ceph-object-store is created without the `zone` section; a realm, zone group, and zone is created with the same name as the ceph-object-store.

Since it is the only ceph-object-store in the realm, the data in the ceph-object-store remain independent and isolated from others on the same cluster.

When a ceph-object-store is created with the `zone` section, the ceph-object-store will join a custom created zone, zone group, and realm each with a different names than its own.

This allows the ceph-object-store to replicate its data over multiple Ceph clusters.

To review core multisite concepts please read the ceph-multisite design overview.

Prerequisites

This guide assumes a Rook cluster as explained in the Quickstart.

Creating Object Multisite

If an admin wants to set up multisite on a Rook Ceph cluster, the admin should create:

1. A realm
2. A zonegroup
3. A zone
4. An object-store with the `zone` section

object-multisite.yaml in the examples directory can be used to create the multisite CRDs.

```
1. kubectl create -f object-multisite.yaml
```

The first zone group created in a realm is the master zone group. The first zone created in a zone group is the master zone.

When a non-master zone or non-master zone group is created, the zone group or zone is not in the Ceph Radosgw Multisite Period until an object-store is created in that zone (and zone group).

The zone will create the pools for the object-store(s) that are in the zone to use.

When one of the multisite CRs (realm, zone group, zone) is deleted the underlying ceph realm/zone group/zone is not deleted, neither are the pools created by the zone. See the “Multisite Cleanup” section for more information.

For more information on the multisite CRDs please read ceph-object-multisite-crd.

Pulling a Realm

If an admin wants to sync data from another cluster, the admin needs to pull a realm on a Rook Ceph cluster from another Rook Ceph (or Ceph) cluster.

To begin doing this, the admin needs 2 pieces of information:

1. An endpoint from the realm being pulled from
2. The access key and the system key of the system user from the realm being pulled from.

Getting the Pull Endpoint

To pull a Ceph realm from a remote Ceph cluster, an `endpoint` must be added to the CephObjectRealm’s `pull` section in the `spec`. This endpoint must be from the master zone in the master zone group of that realm.

If an admin does not know of an endpoint that fits this critera, the admin can find such an endpoint on the remote Ceph cluster (via the tool box if it is a Rook Ceph Cluster) by runnning:

```
radosgw-admin zonegroup get --rgw-realm=$REALM_NAME --rgw-zonegroup=$MASTER_ZONEGROUP_NAME
{
    ...
    "endpoints": [http://10.17.159.77:80],
    ...
}
```

A list of endpoints in the master zone group in the master zone is in the `endpoints` section of the JSON output of the `zonegoup get` command.

This endpoint must also be resolvable from the new Rook Ceph cluster. To test this run the `curl` command on the endpoint:

```
curl -L http://10.17.159.77:80
<?xml version="1.0" encoding="UTF-8"?><ListAllMyBucketsResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
<Owner><ID>anonymous</ID><DisplayName></DisplayName></Owner><Buckets></Buckets></ListAllMyBucketsResult>
```

Finally add the endpoint to the `pull` section of the CephObjectRealm’s spec. The

CephObjectRealm should have the same name as the CephObjectRealm/Ceph realm it is pulling from.

```
apiVersion: ceph.rook.io/v1
kind: CephObjectRealm
metadata:
  name: realm-a
  namespace: rook-ceph
spec:
  pull:
    endpoint: http://10.17.159.77:80
```

Getting Realm Access Key and Secret Key

The access key and secret key of the system user are keys that allow other Ceph clusters to pull the realm of the system user.

Getting the Realm Access Key and Secret Key from the Rook Ceph Cluster

When an admin creates a ceph-object-realm a system user automatically gets created for the realm with an access key and a secret key.

This system user has the name “$REALM_NAME-system-user”. For the example realm, the uid for the system user is “realm-a-system-user”.

These keys for the user are exported as a kubernetes secret called “$REALM_NAME-keys” (ex: realm-a-keys).

To get these keys from the cluster the realm was originally created on, run:

```
kubectl -n $ORIGINAL_CLUSTER_NAMESPACE get secrets realm-a-keys -o yaml > realm-a-keys.yaml
```

Edit the `realm-a-keys.yaml` file, and change the `namespace` with the namespace that the new Rook Ceph cluster exists in.

Then create a kubernetes secret on the pulling Rook Ceph cluster with the same secrets yaml file.

```
kubectl create -f realm-a-keys.yaml
```

Getting the Realm Access Key and Secret Key from a Non Rook Ceph Cluster

The access key and the secret key of the system user can be found in the output of

running the following command on a non-rook ceph cluster:

```
radosgw-admin user info --uid="realm-a-system-user"
{
    ...
    "keys": [
        {
            "user": "realm-a-system-user"
            "access_key": "aSw4blZIKV9nKEU5VC0="
            "secret_key": "JSlDXFt5TlgjSV9QOE9XUndrLiI5JEo9YDBsJg==",
        }
    ],
    ...
}
```

Then base64 encode the each of the keys and create a `.yaml` file for the Kubernetes secret from the following template.

Only the `access-key`, `secret-key`, and `namespace` sections need to be replaced.

```
apiVersion: v1
data:
  access-key: YVN3NGJsWklLVjluS0VVNVZDMD0=
  secret-key: SlNsRFhGdDVUbGdqU1Y5UU9FOVhVbmRyTGlJNUpFbzlZREJzSmc9PQ==
kind: Secret
metadata:
  name: realm-a-keys
  namespace: $NEW_ROOK_CLUSTER_NAMESPACE
type: kubernetes.io/rook
```

Finally, create a kubernetes secret on the pulling Rook Ceph cluster with the new secrets yaml file.

```
kubectl create -f realm-a-keys.yaml
```

Pulling a Realm on a New Rook Ceph Cluster

Once the admin knows the endpoint and the secret for the keys has been created, the admin should create:

1. A CephObjectRealm matching to the realm on the other Ceph cluster, with an endpoint as described above.
2. A CephObjectZoneGroup matching the master zone group name or the master CephObjectZoneGroup from the cluster the the realm was pulled from.
3. A CephObjectZone referring to the CephObjectZoneGroup created above.
4. A CephObjectStore referring to the new CephObjectZone resource.

object-multisite-pull-realm.yaml (with changes) in the examples directory can be used

to create the multisite CRDs.

```
kubectl create -f object-multisite-pull-realm.yaml
```

Multisite Cleanup

Multisite configuration must be cleaned up by hand. Deleting a realm/zone group/zone CR will not delete the underlying Ceph realm, zone group, zone, or the pools associated with a zone.

Realm Deletion

Changes made to the resource's configuration or deletion of the resource are not reflected on the Ceph cluster.

When the ceph-object-realm resource is deleted or modified, the realm is not deleted from the Ceph cluster. Realm deletion must be done via the toolbox.

Deleting a Realm

The Rook toolbox can modify the Ceph Multisite state via the radosgw-admin command.

The following command, run via the toolbox, deletes the realm.

```
radosgw-admin realm rm --rgw-realm=realm-a
```

Zone Group Deletion

Changes made to the resource's configuration or deletion of the resource are not reflected on the Ceph cluster.

When the ceph-object-zone group resource is deleted or modified, the zone group is not deleted from the Ceph cluster. Zone Group deletion must be done through the toolbox.

Deleting a Zone Group

The Rook toolbox can modify the Ceph Multisite state via the radosgw-admin command.

The following command, run via the toolbox, deletes the zone group.

```
radosgw-admin zonegroup delete --rgw-realm=realm-a --rgw-zonegroup=zone-group-a
radosgw-admin period update --commit --rgw-realm=realm-a --rgw-zone-group=zone-group-a
```

Deleting and Reconfiguring the Ceph Object Zone

Changes made to the resource's configuration or deletion of the resource are not reflected on the Ceph cluster.

When the ceph-object-zone resource is deleted or modified, the zone is not deleted from the Ceph cluster. Zone deletion must be done through the toolbox.

Changing the Master Zone

The Rook toolbox can change the master zone in a zone group.

```
radosgw-admin zone modify --rgw-realm=realm-a --rgw-zonegroup=zone-group-a --rgw-zone=zone-a --master
radosgw-admin zonegroup modify --rgw-realm=realm-a --rgw-zonegroup=zone-group-a --master
radosgw-admin period update --commit --rgw-realm=realm-a --rgw-zonegroup=zone-group-a --rgw-zone=zone-a
```

Deleting Zone

The Rook toolbox can modify the Ceph Multisite state via the radosgw-admin command.

There are two scenarios possible when deleting a zone. The following commands, run via the toolbox, deletes the zone if there is only one zone in the zone group.

```
radosgw-admin zone rm --rgw-realm=realm-a --rgw-zone-group=zone-group-a --rgw-zone=zone-a
radosgw-admin period update --commit --rgw-realm=realm-a --rgw-zone-group=zone-group-a --rgw-zone=zone-a
```

In the other scenario, there are more than one zones in a zone group.

Care must be taken when changing which zone is the master zone.

Please read the following documentation before running the below commands:

The following commands, run via toolboxes, remove the zone from the zone group first, then delete the zone.

```
radosgw-admin zonegroup rm --rgw-realm=realm-a --rgw-zone-group=zone-group-a --rgw-zone=zone-a
radosgw-admin period update --commit --rgw-realm=realm-a --rgw-zone-group=zone-group-a --rgw-zone=zone-a
radosgw-admin zone rm --rgw-realm=realm-a --rgw-zone-group=zone-group-a --rgw-zone=zone-a
radosgw-admin period update --commit --rgw-realm=realm-a --rgw-zone-group=zone-group-a --rgw-zone=zone-a
```

When a zone is deleted, the pools for that zone are not deleted.

Deleting Pools for a Zone

The Rook toolbox can delete pools. Deleting pools should be done with caution.

The following documentation on pools should be read before deleting any pools.

When a zone is created the following pools are created for each zone:

```
1. $ZONE_NAME.rgw.control
2. $ZONE_NAME.rgw.meta
3. $ZONE_NAME.rgw.log
4. $ZONE_NAME.rgw.buckets.index
5. $ZONE_NAME.rgw.buckets.non-ec
6. $ZONE_NAME.rgw.buckets.data
```

Here is an example command to delete the .rgw.buckets.data pool for zone-a.

```
1. ceph osd pool rm zone-a.rgw.buckets.data zone-a.rgw.buckets.data --yes-i-really-really-mean-it
```

In this command the pool name **must** be mentioned twice for the pool to be removed.

Removing an Object Store from a Zone

When an object-store (created in a zone) is deleted, the endpoint for that object store is removed from that zone, via

```
1. kubectl delete -f object-store.yaml
```

Removing object store(s) from the master zone of the master zone group should be done with caution. When all of theses object-stores are deleted the period cannot be updated and that realm cannot be pulled.

Shared Filesystem

A shared filesystem can be mounted with read/write permission from multiple pods. This may be useful for applications which can be clustered using a shared filesystem.

This example runs a shared filesystem for the kube-registry.

Prerequisites

This guide assumes you have created a Rook cluster as explained in the main Kubernetes guide

Multiple Filesystems Not Supported

By default only one shared filesystem can be created with Rook. Multiple filesystem support in Ceph is still considered experimental and can be enabled with the environment variable `ROOK_ALLOW_MULTIPLE_FILESYSTEMS` defined in `operator.yaml` .

Please refer to cephfs experimental features page for more information.

Create the Filesystem

Create the filesystem by specifying the desired settings for the metadata pool, data pools, and metadata server in the `CephFilesystem` CRD. In this example we create the metadata pool with replication of three and a single data pool with replication of three. For more options, see the documentation on creating shared filesystems.

Save this shared filesystem definition as `filesystem.yaml` :

```
1.  apiVersion: ceph.rook.io/v1
2.  kind: CephFilesystem
3.  metadata:
4.    name: myfs
5.    namespace: rook-ceph
6.  spec:
7.    metadataPool:
8.      replicated:
9.        size: 3
10.   dataPools:
11.     - replicated:
12.         size: 3
13.   preservePoolsOnDelete: true
14.   metadataServer:
15.     activeCount: 1
16.     activeStandby: true
```

The Rook operator will create all the pools and other resources necessary to start the service. This may take a minute to complete.

```
## Create the filesystem
$ kubectl create -f filesystem.yaml
[...]
## To confirm the filesystem is configured, wait for the mds pods to start
$ kubectl -n rook-ceph get pod -l app=rook-ceph-mds
NAME                                    READY     STATUS    RESTARTS   AGE
rook-ceph-mds-myfs-7d59fdfcf4-h8kw9     1/1       Running   0          12s
rook-ceph-mds-myfs-7d59fdfcf4-kgkjp     1/1       Running   0          12s
```

To see detailed status of the filesystem, start and connect to the Rook toolbox. A new line will be shown with `ceph status` for the `mds` service. In this example, there is one active instance of MDS which is up, with one MDS instance in `standby-replay` mode in case of failover.

```
$ ceph status
  ...
  services:
    mds: myfs-1/1/1 up {[myfs:0]=mzw58b=up:active}, 1 up:standby-replay
```

Provision Storage

Before Rook can start provisioning storage, a StorageClass needs to be created based on the filesystem. This is needed for Kubernetes to interoperate with the CSI driver to create persistent volumes.

> **NOTE**: This example uses the CSI driver, which is the preferred driver going forward for K8s 1.13 and newer. Examples are found in the CSI CephFS directory. For an example of a volume using the flex driver (required for K8s 1.12 and earlier), see the Flex Driver section below.

Save this storage class definition as `storageclass.yaml`:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: rook-cephfs
# Change "rook-ceph" provisioner prefix to match the operator namespace if needed
provisioner: rook-ceph.cephfs.csi.ceph.com
parameters:
  # clusterID is the namespace where operator is deployed.
  clusterID: rook-ceph

  # CephFS filesystem name into which the volume shall be created
  fsName: myfs

  # Ceph pool into which the volume shall be created
  # Required for provisionVolume: "true"
```

```
  pool: myfs-data0

  # Root path of an existing CephFS volume
  # Required for provisionVolume: "false"
  # rootPath: /absolute/path

  # The secrets contain Ceph admin credentials. These are generated automatically by the operator
  # in the same namespace as the cluster.
  csi.storage.k8s.io/provisioner-secret-name: rook-csi-cephfs-provisioner
  csi.storage.k8s.io/provisioner-secret-namespace: rook-ceph
  csi.storage.k8s.io/controller-expand-secret-name: rook-csi-cephfs-provisioner
  csi.storage.k8s.io/controller-expand-secret-namespace: rook-ceph
  csi.storage.k8s.io/node-stage-secret-name: rook-csi-cephfs-node
  csi.storage.k8s.io/node-stage-secret-namespace: rook-ceph

reclaimPolicy: Delete
```

If you've deployed the Rook operator in a namespace other than "rook-ceph" as is common change the prefix in the provisioner to match the namespace you used. For example, if the Rook operator is running in "rook-op" the provisioner value should be "rook-op.rbd.csi.ceph.com".

Create the storage class.

```
kubectl create -f cluster/examples/kubernetes/ceph/csi/cephfs/storageclass.yaml
```

Quotas

> **IMPORTANT**: The CephFS CSI driver uses quotas to enforce the PVC size requested. Only newer kernels support CephFS quotas (kernel version of at least 4.17). If you require quotas to be enforced and the kernel driver does not support it, you can disable the kernel driver and use the FUSE client. This can be done by setting `CSI_FORCE_CEPHFS_KERNEL_CLIENT: false` in the operator deployment (`operator.yaml`). However, it is important to know that when the FUSE client is enabled, there is an issue that during upgrade the application pods will be disconnected from the mount and will need to be restarted. See the upgrade guide for more details.

Consume the Shared Filesystem: K8s Registry Sample

As an example, we will start the kube-registry pod with the shared filesystem as the backing store. Save the following spec as `kube-registry.yaml` :

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: cephfs-pvc
spec:
  accessModes:
  - ReadWriteMany
```

```
  resources:
    requests:
      storage: 1Gi
  storageClassName: rook-cephfs
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kube-registry
  namespace: kube-system
  labels:
    k8s-app: kube-registry
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 3
  selector:
    matchLabels:
      k8s-app: kube-registry
  template:
    metadata:
      labels:
        k8s-app: kube-registry
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
      - name: registry
        image: registry:2
        imagePullPolicy: Always
        resources:
          limits:
            cpu: 100m
            memory: 100Mi
        env:
        # Configuration reference: https://docs.docker.com/registry/configuration/
        - name: REGISTRY_HTTP_ADDR
          value: :5000
        - name: REGISTRY_HTTP_SECRET
          value: "Ple4seCh4ngeThisN0tAVerySecretV4lue"
        - name: REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY
          value: /var/lib/registry
        volumeMounts:
        - name: image-store
          mountPath: /var/lib/registry
        ports:
        - containerPort: 5000
          name: registry
          protocol: TCP
        livenessProbe:
          httpGet:
            path: /
            port: registry
        readinessProbe:
          httpGet:
```

```
            path: /
            port: registry
      volumes:
      - name: image-store
        persistentVolumeClaim:
          claimName: cephfs-pvc
          readOnly: false
```

Create the Kube registry deployment:

```
kubectl create -f cluster/examples/kubernetes/ceph/csi/cephfs/kube-registry.yaml
```

You now have a docker registry which is HA with persistent storage.

Kernel Version Requirement

If the Rook cluster has more than one filesystem and the application pod is scheduled to a node with kernel version older than 4.7, inconsistent results may arise since kernels older than 4.7 do not support specifying filesystem namespaces.

Consume the Shared Filesystem: Toolbox

Once you have pushed an image to the registry (see the instructions to expose and use the kube-registry), verify that kube-registry is using the filesystem that was configured above by mounting the shared filesystem in the toolbox pod. See the Direct Filesystem topic for more details.

Teardown

To clean up all the artifacts created by the filesystem demo:

```
kubectl delete -f kube-registry.yaml
```

To delete the filesystem components and backing data, delete the Filesystem CRD.

> WARNING: Data will be deleted if preservePoolsOnDelete=false.

```
kubectl -n rook-ceph delete cephfilesystem myfs
```

Note: If the “preservePoolsOnDelete” filesystem attribute is set to true, the above command won’t delete the pools. Creating again the filesystem with the same CRD will reuse again the previous pools.

Flex Driver

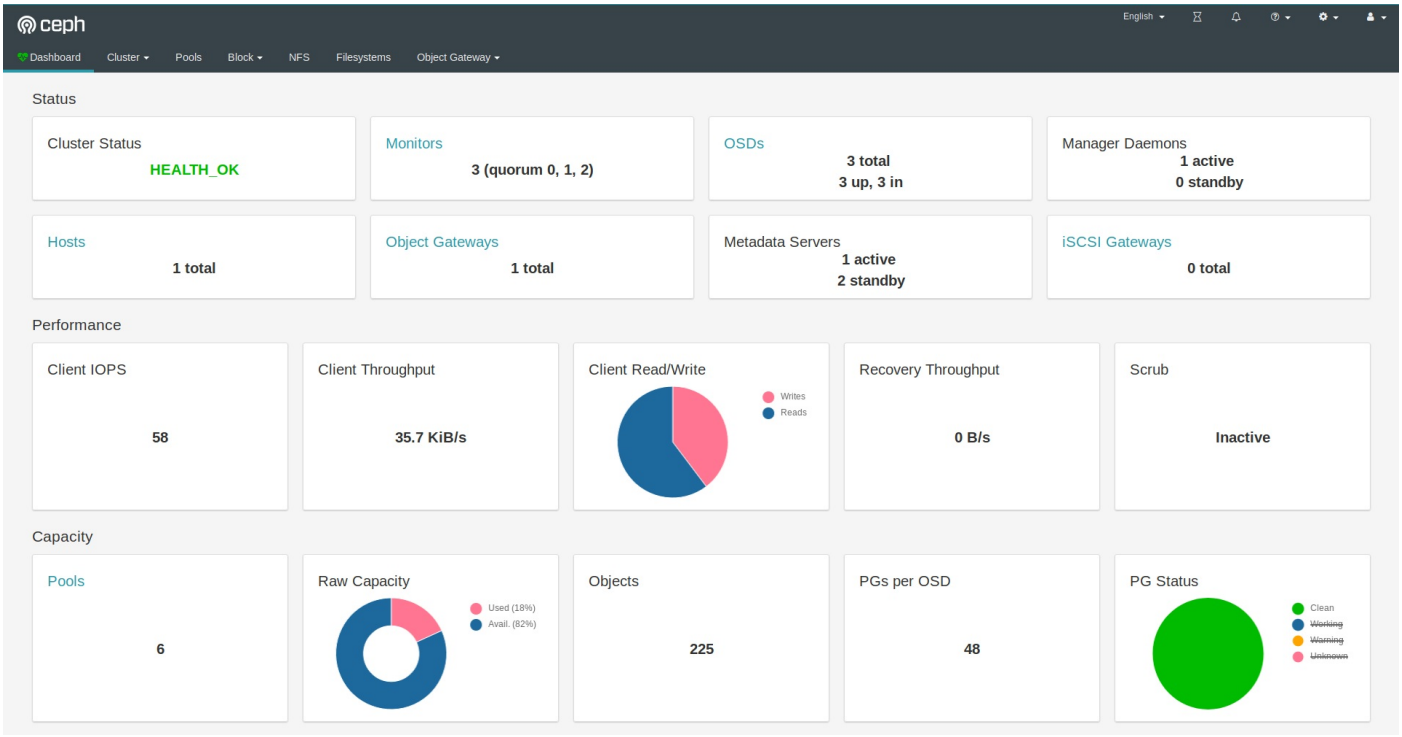
To create a volume based on the flex driver instead of the CSI driver, see the kube-registry.yaml example manifest or refer to the complete flow in the Rook v1.0 Shared Filesystem documentation.

Advanced Example: Erasure Coded Filesystem

The Ceph filesystem example can be found here: Ceph Shared Filesystem - Samples - Erasure Coded.

Ceph Dashboard

The dashboard is a very helpful tool to give you an overview of the status of your Ceph cluster, including overall health, status of the mon quorum, status of the mgr, osd, and other Ceph daemons, view pools and PG status, show logs for the daemons, and more. Rook makes it simple to enable the dashboard.



Enable the Ceph Dashboard

The dashboard can be enabled with settings in the CephCluster CRD. The CephCluster CRD must have the dashboard `enabled` setting set to `true`. This is the default setting in the example manifests.

```
1.  spec:
2.    dashboard:
3.      enabled: true
```

The Rook operator will enable the ceph-mgr dashboard module. A service object will be created to expose that port inside the Kubernetes cluster. Rook will enable port 8443 for https access.

This example shows that port 8443 was configured.

```
1. kubectl -n rook-ceph get service
2. NAME                      TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
3. rook-ceph-mgr             ClusterIP   10.108.111.192   <none>        9283/TCP         3h
4. rook-ceph-mgr-dashboard   ClusterIP   10.110.113.240   <none>        8443/TCP         3h
```

The first service is for reporting the Prometheus metrics, while the latter service is for the dashboard. If you are on a node in the cluster, you will be able to connect to the dashboard by using either the DNS name of the service at `https://rook-ceph-mgr-dashboard-https:8443` or by connecting to the cluster IP, in this example at `https://10.110.113.240:8443`.

> **IMPORTANT:** Please note the dashboard will only be enabled for the first Ceph object store created by Rook.

Login Credentials

After you connect to the dashboard you will need to login for secure access. Rook creates a default user named `admin` and generates a secret called `rook-ceph-dashboard-admin-password` in the namespace where the Rook Ceph cluster is running. To retrieve the generated password, you can run the following:

```
    kubectl -n rook-ceph get secret rook-ceph-dashboard-password -o jsonpath="{['data']['password']}" | base64 --
1.  decode && echo
```

Configure the Dashboard

The following dashboard configuration settings are supported:

```
1.  spec:
2.    dashboard:
3.      urlPrefix: /ceph-dashboard
4.      port: 8443
5.      ssl: true
```

- `urlPrefix` If you are accessing the dashboard via a reverse proxy, you may wish to serve it under a URL prefix. To get the dashboard to use hyperlinks that include your prefix, you can set the `urlPrefix` setting.
- `port` The port that the dashboard is served on may be changed from the default using the `port` setting. The corresponding K8s service exposing the port will automatically be updated.
- `ssl` The dashboard may be served without SSL (useful for when you deploy the dashboard behind a proxy already served using SSL) by setting the `ssl` option to be false.

Viewing the Dashboard External to the Cluster

Commonly you will want to view the dashboard from outside the cluster. For example, on a development machine with the cluster running inside minikube you will want to access the dashboard from the host.

There are several ways to expose a service that will depend on the environment you are running in. You can use an Ingress Controller or other methods for exposing services such as NodePort, LoadBalancer, or ExternalIPs.

Node Port

The simplest way to expose the service in minikube or similar environment is using the NodePort to open a port on the VM that can be accessed by the host. To create a service with the NodePort, save this yaml as `dashboard-external-https.yaml`.

```
1. apiVersion: v1
2. kind: Service
3. metadata:
4.   name: rook-ceph-mgr-dashboard-external-https
5.   namespace: rook-ceph
6.   labels:
7.     app: rook-ceph-mgr
8.     rook_cluster: rook-ceph
9. spec:
10.   ports:
11.   - name: dashboard
12.     port: 8443
13.     protocol: TCP
14.     targetPort: 8443
15.   selector:
16.     app: rook-ceph-mgr
17.     rook_cluster: rook-ceph
18.   sessionAffinity: None
19.   type: NodePort
```

Now create the service:

```
1. kubectl create -f dashboard-external-https.yaml
```

You will see the new service `rook-ceph-mgr-dashboard-external-https` created:

```
1. $ kubectl -n rook-ceph get service
2. NAME                                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
3. rook-ceph-mgr                        ClusterIP   10.108.111.192   <none>        9283/TCP         4h
4. rook-ceph-mgr-dashboard              ClusterIP   10.110.113.240   <none>        8443/TCP         4h
5. rook-ceph-mgr-dashboard-external-https NodePort  10.101.209.6     <none>        8443:31176/TCP   4h
```

In this example, port `31176` will be opened to expose port `8443` from the ceph-mgr pod. Find the ip address of the VM. If using minikube, you can run `minikube ip` to find the ip address. Now you can enter the URL in your browser such as `https://192.168.99.110:31176` and the dashboard will appear.

Load Balancer

If you have a cluster on a cloud provider that supports load balancers, you can create a service that is provisioned with a public hostname. The yaml is the same as `dashboard-external-https.yaml` except for the following property:

```
spec:
[...]
  type: LoadBalancer
```

Now create the service:

```
kubectl create -f dashboard-loadbalancer.yaml
```

You will see the new service `rook-ceph-mgr-dashboard-loadbalancer` created:

```
$ kubectl -n rook-ceph get service
NAME                                     TYPE           CLUSTER-IP       EXTERNAL-IP
PORT(S)             AGE
rook-ceph-mgr                            ClusterIP      172.30.11.40     <none>
9283/TCP            4h
rook-ceph-mgr-dashboard                  ClusterIP      172.30.203.185   <none>
8443/TCP            4h
rook-ceph-mgr-dashboard-loadbalancer     LoadBalancer   172.30.27.242    a7f23e8e2839511e9b7a5122b08f2038-
1251669398.us-east-1.elb.amazonaws.com   8443:32747/TCP      4h
```

Now you can enter the URL in your browser such as `https://a7f23e8e2839511e9b7a5122b08f2038-1251669398.us-east-1.elb.amazonaws.com:8443` and the dashboard will appear.

Ingress Controller

If you have a cluster with an nginx Ingress Controller and a Certificate Manager (e.g. cert-manager) then you can create an Ingress like the one below. This example achieves four things:

1. Exposes the dashboard on the Internet (using an reverse proxy)
2. Issues an valid TLS Certificate for the specified domain name (using ACME)
3. Tells the reverse proxy that the dashboard itself uses HTTPS
4. Tells the reverse proxy that the dashboard itself does not have a valid certificate (it is self-signed)

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: rook-ceph-mgr-dashboard
  namespace: rook-ceph
  annotations:
    kubernetes.io/ingress.class: "nginx"
    kubernetes.io/tls-acme: "true"
    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
    nginx.ingress.kubernetes.io/server-snippet: |
```

```
      proxy_ssl_verify off;
spec:
  tls:
   - hosts:
     - rook-ceph.example.com
     secretName: rook-ceph.example.com
  rules:
  - host: rook-ceph.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: rook-ceph-mgr-dashboard
          servicePort: https-dashboard
```

Customise the Ingress resource to match your cluster. Replace the example domain name `rook-ceph.example.com` with a domain name that will resolve to your Ingress Controller (creating the DNS entry if required).

Now create the Ingress:

```
kubectl create -f dashboard-ingress-https.yaml
```

You will see the new Ingress `rook-ceph-mgr-dashboard` created:

```
$ kubectl -n rook-ceph get ingress
NAME                      HOSTS                     ADDRESS   PORTS     AGE
rook-ceph-mgr-dashboard   rook-ceph.example.com     80, 443   5m
```

And the new Secret for the TLS certificate:

```
$ kubectl -n rook-ceph get secret rook-ceph.example.com
NAME                      TYPE                DATA      AGE
rook-ceph.example.com     kubernetes.io/tls   2         4m
```

You can now browse to `https://rook-ceph.example.com/` to log into the dashboard.

Prometheus Monitoring

Each Rook Ceph cluster has some built in metrics collectors/exporters for monitoring with Prometheus.

If you do not have Prometheus running, follow the steps below to enable monitoring of Rook. If your cluster already contains a Prometheus instance, it will automatically discover Rooks scrape endpoint using the standard `prometheus.io/scrape` and `prometheus.io/port` annotations.

> **NOTE**: This assumes that the Prometheus instances is searching all your Kubernetes namespaces for Pods with these annotations.

Prometheus Operator

First the Prometheus operator needs to be started in the cluster so it can watch for our requests to start monitoring Rook and respond by deploying the correct Prometheus pods and configuration. A full explanation can be found in the Prometheus operator repository on GitHub, but the quick instructions can be found here:

```
1. kubectl apply -f https://raw.githubusercontent.com/coreos/prometheus-operator/v0.40.0/bundle.yaml
```

This will start the Prometheus operator, but before moving on, wait until the operator is in the `Running` state:

```
1. kubectl get pod
```

Once the Prometheus operator is in the `Running` state, proceed to the next section to create a Prometheus instance.

Prometheus Instances

With the Prometheus operator running, we can create a service monitor that will watch the Rook cluster and collect metrics regularly. From the root of your locally cloned Rook repo, go the monitoring directory:

```
1. git clone --single-branch --branch release-1.4 https://github.com/rook/rook.git
2. cd rook/cluster/examples/kubernetes/ceph/monitoring
```

Create the service monitor as well as the Prometheus server pod and service:

```
1. kubectl create -f service-monitor.yaml
2. kubectl create -f prometheus.yaml
```

```
3. kubectl create -f prometheus-service.yaml
```

Ensure that the Prometheus server pod gets created and advances to the `Running` state before moving on:

```
1. kubectl -n rook-ceph get pod prometheus-rook-prometheus-0
```

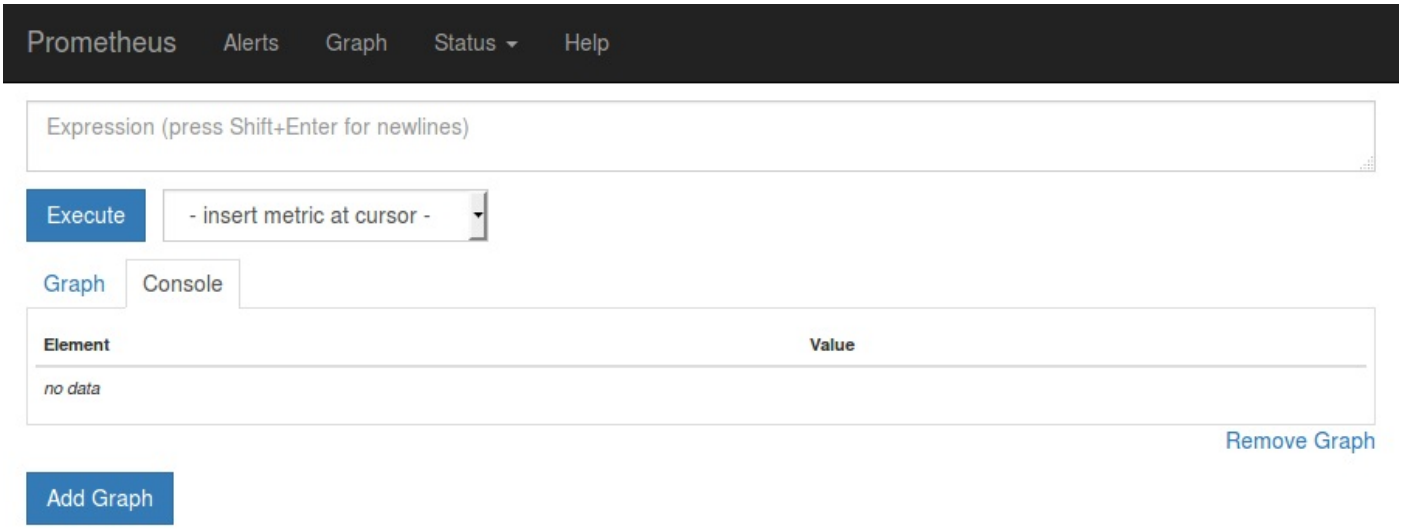
> **NOTE**: It is not recommended to consume storage from the Ceph cluster for Prometheus. If the Ceph cluster fails, Prometheus would become unresponsive and thus not alert you of the failure.

Prometheus Web Console

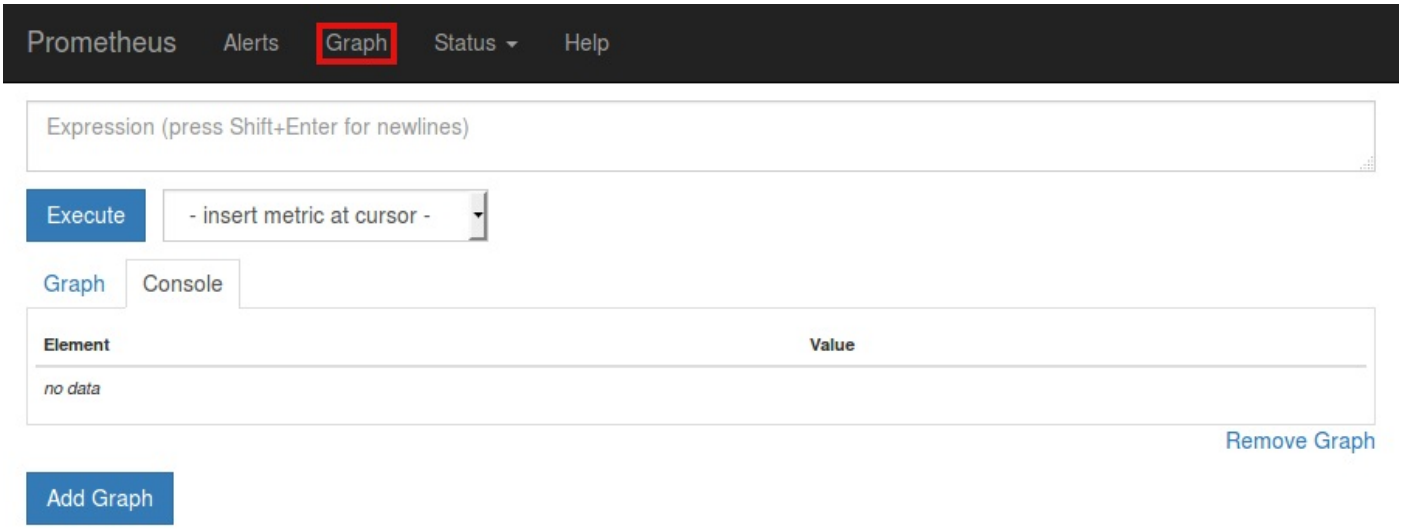
Once the Prometheus server is running, you can open a web browser and go to the URL that is output from this command:

```
1. echo "http://$(kubectl -n rook-ceph -o jsonpath={.status.hostIP} get pod prometheus-rook-prometheus-0):30900"
```

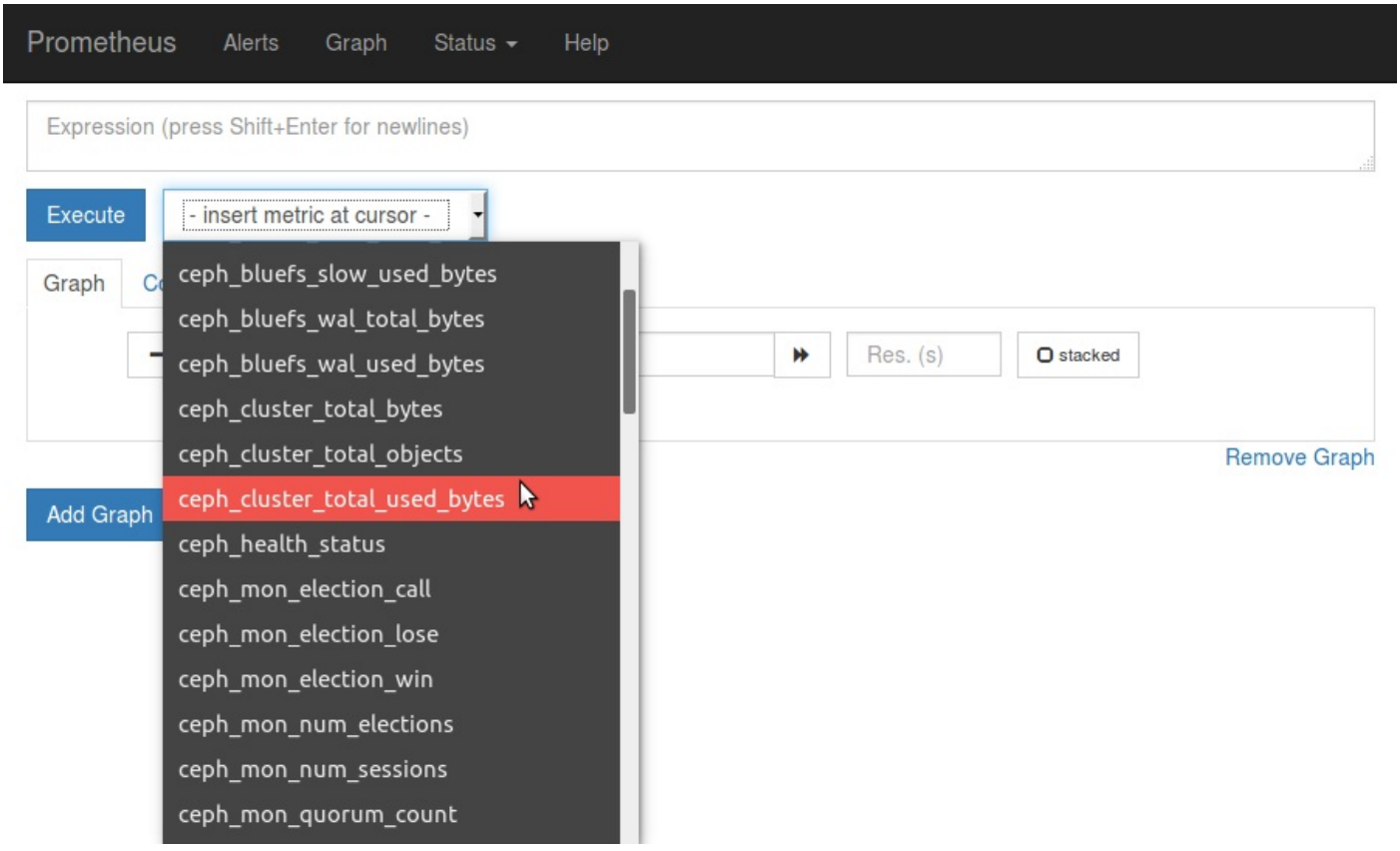
You should now see the Prometheus monitoring website.



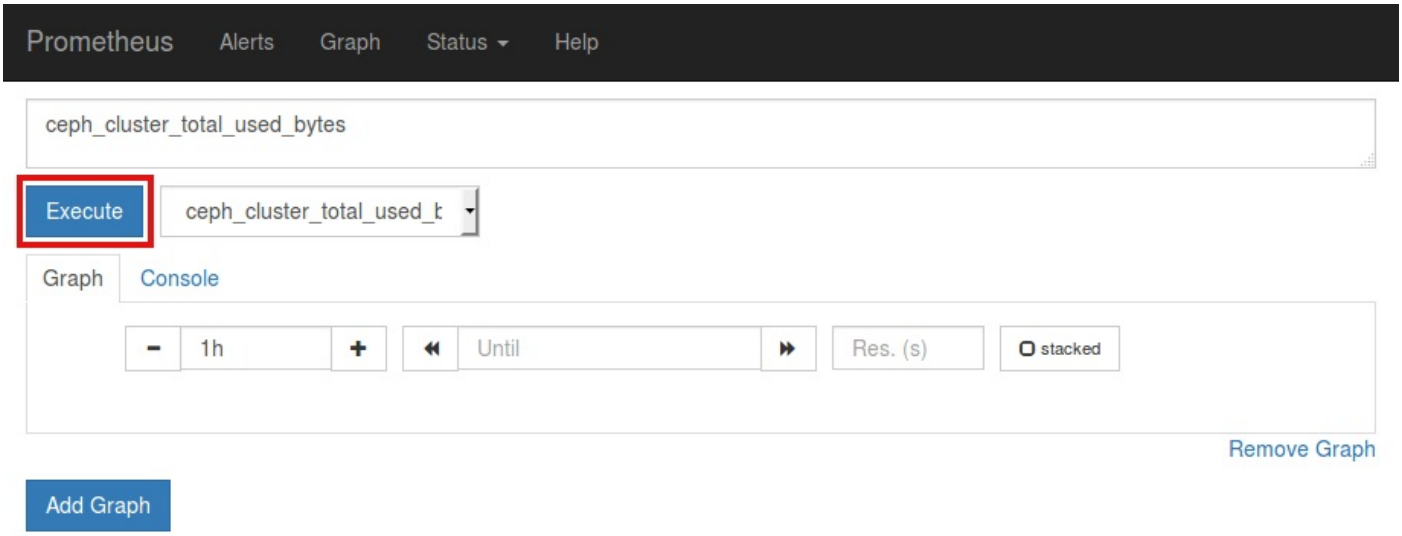
Click on `Graph` in the top navigation bar.



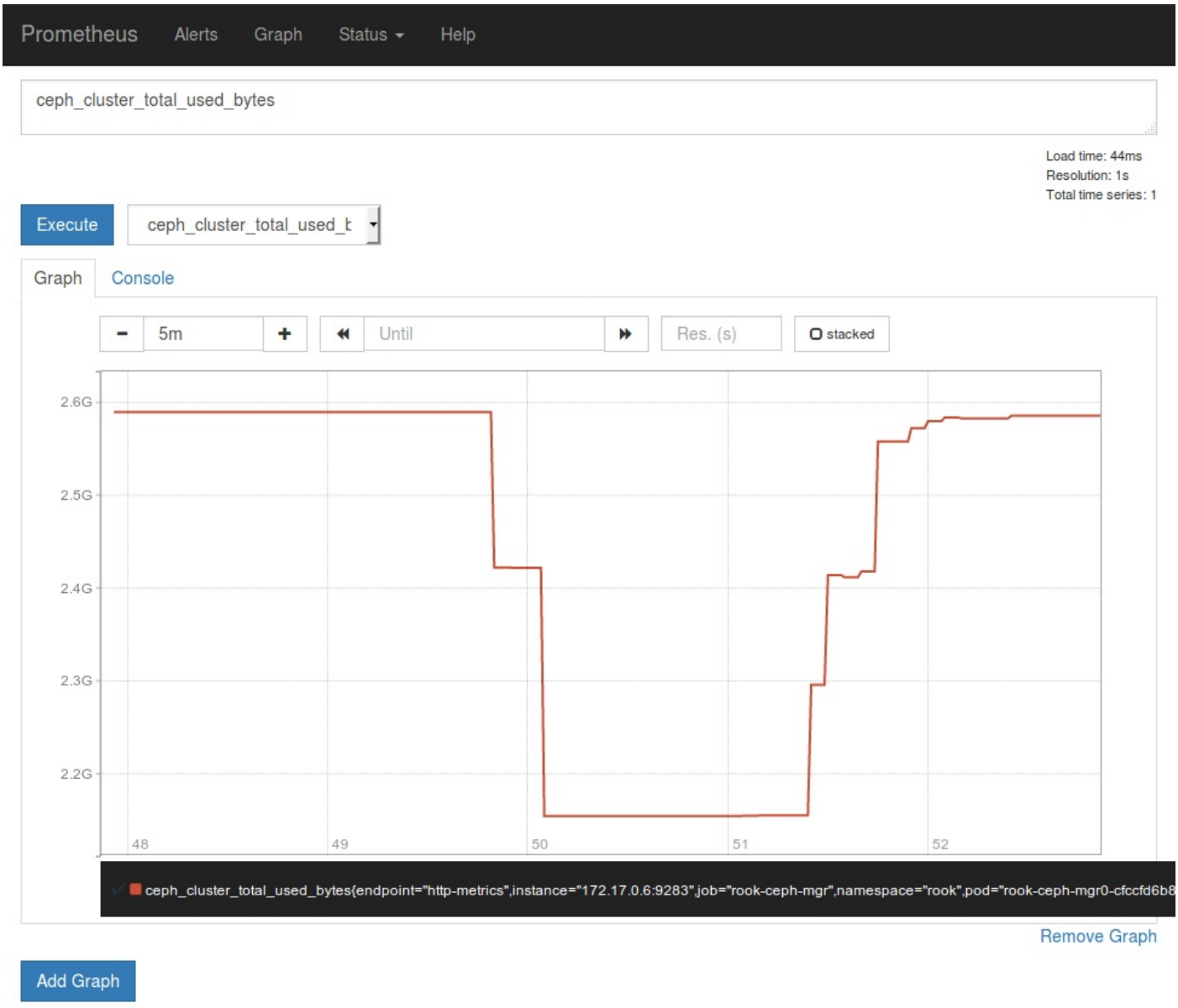
In the dropdown that says `insert metric at cursor`, select any metric you would like to see, for example `ceph_cluster_total_used_bytes`



Click on the `Execute` button.



Below the `Execute` button, ensure the `Graph` tab is selected and you should now see a graph of your chosen metric over time.



Prometheus Consoles

You can find Prometheus Consoles for and from Ceph here: GitHub ceph/cephmetrics - dashboards/current directory.

A guide to how you can write your own Prometheus consoles can be found on the official Prometheus site here: Prometheus.io Documentation - Console Templates.

Prometheus Alerts

To enable the Ceph Prometheus alerts follow these steps:

1. Create the RBAC rules to enable monitoring.

```
1. kubectl create -f cluster/examples/kubernetes/ceph/monitoring/rbac.yaml
```

1. Make following changes to your CephCluster object (e.g., `cluster.yaml`).

```
1. apiVersion: ceph.rook.io/v1
2. kind: CephCluster
3. metadata:
4.   name: rook-ceph
5.   namespace: rook-ceph
6. [...]
7. spec:
8. [...]
9.   monitoring:
10.     enabled: true
11.     rulesNamespace: "rook-ceph"
12. [...]
```

(Where `rook-ceph` is the CephCluster name / namespace)

1. Deploy or update the CephCluster object.

```
1. kubectl apply -f cluster.yaml
```

> **NOTE**: This expects the Prometheus Operator and a Prometheus instance to be pre-installed by the admin.

Grafana Dashboards

The dashboards have been created by @galexrt. For feedback on the dashboards please reach out to him on the Rook.io Slack.

> **NOTE**: The dashboards are only compatible with Grafana 5.0.3 or higher.
>
> Also note that the dashboards are updated from time to time, to fix issues and improve them.

The following Grafana dashboards are available:

- Ceph - Cluster
- Ceph - OSD
- Ceph - Pools

Teardown

To clean up all the artifacts created by the monitoring walk-through, copy/paste the entire block below (note that errors about resources “not found” can be ignored):

```
kubectl delete -f service-monitor.yaml
kubectl delete -f prometheus.yaml
kubectl delete -f prometheus-service.yaml
kubectl delete -f https://raw.githubusercontent.com/coreos/prometheus-operator/v0.40.0/bundle.yaml
```

Then the rest of the instructions in the Prometheus Operator docs can be followed to finish cleaning up.

Special Cases

Tectonic Bare Metal

Tectonic strongly discourages the `tectonic-system` Prometheus instance to be used outside their intentions, so you need to create a new Prometheus Operator yourself. After this you only need to create the service monitor as stated above.

CSI Liveness

To integrate CSI liveness and grpc into ceph monitoring we will need to deploy a service and service monitor.

```
kubectl create -f csi-metrics-service-monitor.yaml
```

This will create the service monitor to have promethues monitor CSI

Collecting RBD per-image IO statistics

RBD per-image IO statistics collection is disabled by default. This can be enabled by setting `enableRBDStats: true` in the CephBlockPool spec. Prometheus does not need to be restarted after enabling it.

Ceph Cluster CRD

Rook allows creation and customization of storage clusters through the custom resource definitions (CRDs). There are two different modes to create your cluster, depending on whether storage can be dynamically provisioned on which to base the Ceph cluster.

1. Specify host paths and raw devices
2. Specify the storage class Rook should use to consume storage via PVCs

Following is an example for each of these approaches.

Host-based Cluster

To get you started, here is a simple example of a CRD to configure a Ceph cluster with all nodes and all devices. Next example is where Mons and OSDs are backed by PVCs. More examples are included later in this doc.

> **NOTE**: In addition to your CephCluster object, you need to create the namespace, service accounts, and RBAC rules for the namespace you are going to create the CephCluster in. These resources are defined in the example `common.yaml` .

```
apiVersion: ceph.rook.io/v1
kind: CephCluster
metadata:
  name: rook-ceph
  namespace: rook-ceph
spec:
  cephVersion:
    # see the "Cluster Settings" section below for more details on which image of ceph to run
    image: ceph/ceph:v15.2.4
  dataDirHostPath: /var/lib/rook
  mon:
    count: 3
    allowMultiplePerNode: true
  storage:
    useAllNodes: true
    useAllDevices: true
```

PVC-based Cluster

> **NOTE**: Kubernetes version 1.13.0 or greater is required to provision OSDs on PVCs.

```
apiVersion: ceph.rook.io/v1
kind: CephCluster
metadata:
  name: rook-ceph
  namespace: rook-ceph
```

```
spec:
  cephVersion:
    # see the "Cluster Settings" section below for more details on which image of ceph to run
    image: ceph/ceph:v15.2.4
  dataDirHostPath: /var/lib/rook
  mon:
    count: 3
    volumeClaimTemplate:
      spec:
        storageClassName: local-storage
        resources:
          requests:
            storage: 10Gi
  storage:
   storageClassDeviceSets:
    - name: set1
      count: 3
      portable: false
      tuneDeviceClass: false
      encrypted: false
      volumeClaimTemplates:
      - metadata:
          name: data
        spec:
          resources:
            requests:
              storage: 10Gi
          # IMPORTANT: Change the storage class depending on your environment (e.g. local-storage, gp2)
          storageClassName: local-storage
          volumeMode: Block
          accessModes:
            - ReadWriteOnce
```

For a more advanced scenario, such as adding a dedicated device you can refer to the dedicated metadata device for OSD on PVC section.

Settings

Settings can be specified at the global level to apply to the cluster as a whole, while other settings can be specified at more fine-grained levels. If any setting is unspecified, a suitable default will be used automatically.

Cluster metadata

- `name`: The name that will be used internally for the Ceph cluster. Most commonly the name is the same as the namespace since multiple clusters are not supported in the same namespace.
- `namespace`: The Kubernetes namespace that will be created for the Rook cluster. The services, pods, and other resources created by the operator will be added to this

namespace. The common scenario is to create a single Rook cluster. If multiple clusters are created, they must not have conflicting devices or host paths.

Cluster Settings

- `external`:
  - `enable`: if `true`, the cluster will not be managed by Rook but via an external entity. This mode is intended to connect to an existing cluster. In this case, Rook will only consume the external cluster. However, Rook will be able to deploy various daemons in Kubernetes such as object gateways, mds and nfs if an image is provided and will refuse otherwise. If this setting is enabled **all** the other options will be ignored except `cephVersion.image` and `dataDirHostPath`. See external cluster configuration. If `cephVersion.image` is left blank, Rook will refuse the creation of extra CRs like object, file and nfs.
- `cephVersion`: The version information for launching the ceph daemons.
  - `image`: The image used for running the ceph daemons. For example, `ceph/ceph:v14.2.10` or `ceph/ceph:v15.2.4`. For more details read the container images section. For the latest ceph images, see the Ceph DockerHub. To ensure a consistent version of the image is running across all nodes in the cluster, it is recommended to use a very specific image version. Tags also exist that would give the latest version, but they are only recommended for test environments. For example, the tag `v14` will be updated each time a new nautilus build is released. Using the `v14` or similar tag is not recommended in production because it may lead to inconsistent versions of the image running across different nodes in the cluster.
  - `allowUnsupported`: If `true`, allow an unsupported major version of the Ceph release. Currently `nautilus` and `octopus` are supported. Future versions such as `pacific` would require this to be set to `true`. Should be set to `false` in production.
- `dataDirHostPath`: The path on the host (hostPath) where config and data should be stored for each of the services. If the directory does not exist, it will be created. Because this directory persists on the host, it will remain after pods are deleted. Following paths and any of their subpaths **must not be used**: `/etc/ceph`, `/rook` or `/var/log/ceph`.
  - On **Minikube** environments, use `/data/rook`. Minikube boots into a tmpfs but it provides some directories where files can be persisted across reboots. Using one of these directories will ensure that Rook's data and configuration files are persisted and that enough storage space is available.
  - **WARNING**: For test scenarios, if you delete a cluster and start a new cluster on the same hosts, the path used by `dataDirHostPath` must be deleted. Otherwise, stale keys and other config will remain from the previous cluster and the new mons will fail to start. If this value is empty, each pod will get an ephemeral directory to store their config files that is tied to the lifetime of the pod running on that node. More details can be found in the Kubernetes empty dir docs.
- `skipUpgradeChecks`: if set to true Rook won't perform any upgrade checks on Ceph

daemons during an upgrade. Use this at **YOUR OWN RISK**, only if you know what you're doing. To understand Rook's upgrade process of Ceph, read the upgrade doc.

- `continueUpgradeAfterChecksEvenIfNotHealthy`: if set to true Rook will continue the OSD daemon upgrade process even if the PGs are not clean, or continue with the MDS upgrade even the file system is not healthy.
- `dashboard`: Settings for the Ceph dashboard. To view the dashboard in your browser see the dashboard guide.
  - `enabled`: Whether to enable the dashboard to view cluster status
  - `urlPrefix`: Allows to serve the dashboard under a subpath (useful when you are accessing the dashboard via a reverse proxy)
  - `port`: Allows to change the default port where the dashboard is served
  - `ssl`: Whether to serve the dashboard via SSL, ignored on Ceph versions older than `13.2.2`
- `monitoring`: Settings for monitoring Ceph using Prometheus. To enable monitoring on your cluster see the monitoring guide.
  - `enabled`: Whether to enable prometheus based monitoring for this cluster
  - `rulesNamespace`: Namespace to deploy prometheusRule. If empty, namespace of the cluster will be used. Recommended:
    - If you have a single Rook Ceph cluster, set the `rulesNamespace` to the same namespace as the cluster or keep it empty.
    - If you have multiple Rook Ceph clusters in the same Kubernetes cluster, choose the same namespace to set `rulesNamespace` for all the clusters (ideally, namespace with prometheus deployed). Otherwise, you will get duplicate alerts with duplicate alert definitions.
- `network`: For the network settings for the cluster, refer to the network configuration settings
- `mon`: contains mon related options mon settings For more details on the mons and when to choose a number other than `3`, see the mon health design doc.
- `mgr`: manager top level section
  - `modules`: is the list of Ceph manager modules to enable
- `crashCollector`: The settings for crash collector daemon(s).
  - `disable`: is set to `true`, the crash collector will not run on any node where a Ceph daemon runs
- `annotations`: annotations configuration settings
- `placement`: placement configuration settings
- `resources`: resources configuration settings
- `priorityClassNames`: priority class names configuration settings
- `storage`: Storage selection and configuration that will be used across the cluster. Note that these settings can be overridden for specific nodes.
  - `useAllNodes`: `true` or `false`, indicating if all nodes in the cluster should be used for storage according to the cluster level storage selection and configuration values. If individual nodes are specified under the `nodes` field, then `useAllNodes` must be set to `false`.
  - `nodes`: Names of individual nodes in the cluster that should have their storage included in accordance with either the cluster level configuration specified above or any node specific overrides described in the next section

below. `useAllNodes` must be set to `false` to use specific nodes and their config. See node settings below.
    - `config`: Config settings applied to all OSDs on the node unless overridden by `devices`. See the config settings below.
    - storage selection settings
    - Storage Class Device Sets
- `disruptionManagement`: The section for configuring management of daemon disruptions
    - `managePodBudgets`: if `true`, the operator will create and manage PodDisruptionBudgets for OSD, Mon, RGW, and MDS daemons. OSD PDBs are managed dynamically via the strategy outlined in the design. The operator will block eviction of OSDs by default and unblock them safely when drains are detected.
    - `osdMaintenanceTimeout`: is a duration in minutes that determines how long an entire failureDomain like `region/zone/host` will be held in `noout` (in addition to the default DOWN/OUT interval) when it is draining. This is only relevant when `managePodBudgets` is `true`. The default value is `30` minutes.
    - `manageMachineDisruptionBudgets`: if `true`, the operator will create and manage MachineDisruptionBudgets to ensure OSDs are only fenced when the cluster is healthy. Only available on OpenShift.
    - `machineDisruptionBudgetNamespace`: the namespace in which to watch the MachineDisruptionBudgets.
- `removeOSDsIfOutAndSafeToRemove`: If `true` the operator will remove the OSDs that are down and whose data has been restored to other OSDs. In Ceph terms, the osds are `out` and `safe-to-destroy` when then would be removed.
- `cleanupPolicy`: cleanup policy settings

Ceph container images

Official releases of Ceph Container images are available from Docker Hub.

These are general purpose Ceph container with all necessary daemons and dependencies installed.

TAG	MEANING
vRELNUM	Latest release in this series (e.g., *v14* = Nautilus)
vRELNUM.Y	Latest stable release in this stable series (e.g., v14.2)
vRELNUM.Y.Z	A specific release (e.g., v14.2.5)
vRELNUM.Y.Z-YYYYMMDD	A specific build (e.g., v14.2.5-20191203)

A specific will contain a specific release of Ceph as well as security fixes from the Operating System.

Mon Settings

- `count`: Set the number of mons to be started. The number should be odd and between `1` and `9`. If not specified the default is set to `3` and `allowMultiplePerNode` is

also set to `true`.
- `allowMultiplePerNode`: Enable (`true`) or disable (`false`) the placement of multiple mons on one node. Default is `false`.
- `volumeClaimTemplate`: A `PersistentVolumeSpec` used by Rook to create PVCs for monitor storage. This field is optional, and when not provided, HostPath volume mounts are used. The current set of fields from template that are used are `storageClassName` and the `storage` resource request and limit. The default storage size request for new PVCs is `10Gi`. Ensure that associated storage class is configured to use `volumeBindingMode: WaitForFirstConsumer`. This setting only applies to new monitors that are created when the requested number of monitors increases, or when a monitor fails and is recreated. An example CRD configuration is provided below.

If these settings are changed in the CRD the operator will update the number of mons during a periodic check of the mon health, which by default is every 45 seconds.

To change the defaults that the operator uses to determine the mon health and whether to failover a mon, refer to the health settings. The intervals should be small enough that you have confidence the mons will maintain quorum, while also being long enough to ignore network blips where mons are failed over too often.

Mgr Settings

You can use the cluster CR to enable or disable any manager module. This can be configured like so:

```
mgr:
  modules:
  - name: <name of the module>
    enabled: true
```

Some modules will have special configuration to ensure the module is fully functional after being enabled. Specifically:

- `pg_autoscaler`: Rook will configure all new pools with PG autoscaling by setting: `osd_pool_default_pg_autoscale_mode = on`

Network Configuration Settings

If not specified, the default SDN will be used. Configure the network that will be enabled for the cluster and services.

- `provider`: Specifies the network provider that will be used to connect the network interface. You can choose between `host`, and `multus`.
- `selectors`: List the network selector(s) that will be used associated by a key.

> **NOTE:** Changing networking configuration after a Ceph cluster has been deployed is NOT supported and will result in a non-functioning cluster.

Host Networking

To use host networking, set `provider: host` .

Multus (EXPERIMENTAL)

Rook has experimental support for Multus.

The selector keys are required to be `public` and `cluster` where each represent:

- `public` : client communications with the cluster (reads/writes)
- `cluster` : internal Ceph replication network

If you want to learn more, please read Ceph Networking reference.

Based on the configuration, the operator will do the following:

1. if only the `public` selector is specified both communication and replication will happen on that network
2. if both `public` and `cluster` selectors are specified the first one will run the communication network and the second the replication network

In order to work, each selector value must match a `NetworkAttachmentDefinition` object name in Multus. For example, you can do:

- `public` : “my-public-storage-network”
- `cluster` : “my-replication-storage-network”

For `multus` network provider, an already working cluster with Multus networking is required. Network attachment definition that later will be attached to the cluster needs to be created before the Cluster CRD. The Network attachment definitions should be using whereabouts cni. If Rook cannot find the provided Network attachment definition it will fail running the Ceph OSD pods. You can add the Multus network attachment selection annotation selecting the created network attachment definition on `selectors` .

Node Settings

In addition to the cluster level settings specified above, each individual node can also specify configuration to override the cluster level settings and defaults. If a node does not specify any configuration then it will inherit the cluster level settings.

- `name` : The name of the node, which should match its `kubernetes.io/hostname` label.
- `config` : Config settings applied to all OSDs on the node unless overridden by `devices` . See the config settings below.
- storage selection settings

When `useAllNodes` is set to `true` , Rook attempts to make Ceph cluster management as

hands-off as possible while still maintaining reasonable data safety. If a usable node comes online, Rook will begin to use it automatically. To maintain a balance between hands-off usability and data safety, Nodes are removed from Ceph as OSD hosts only (1) if the node is deleted from Kubernetes itself or (2) if the node has its taints or affinities modified in such a way that the node is no longer usable by Rook. Any changes to taints or affinities, intentional or unintentional, may affect the data reliability of the Ceph cluster. In order to help protect against this somewhat, deletion of nodes by taint or affinity modifications must be “confirmed” by deleting the Rook-Ceph operator pod and allowing the operator deployment to restart the pod.

For production clusters, we recommend that `useAllNodes` is set to `false` to prevent the Ceph cluster from suffering reduced data reliability unintentionally due to a user mistake. When `useAllNodes` is set to `false`, Rook relies on the user to be explicit about when nodes are added to or removed from the Ceph cluster. Nodes are only added to the Ceph cluster if the node is added to the Ceph cluster resource. Similarly, nodes are only removed if the node is removed from the Ceph cluster resource.

Node Updates

Nodes can be added and removed over time by updating the Cluster CRD, for example with `kubectl -n rook-ceph edit cephcluster rook-ceph`. This will bring up your default text editor and allow you to add and remove storage nodes from the cluster. This feature is only available when `useAllNodes` has been set to `false`.

Storage Selection Settings

Below are the settings available, both at the cluster and individual node level, for selecting which storage resources will be included in the cluster.

- `useAllDevices`: `true` or `false`, indicating whether all devices found on nodes in the cluster should be automatically consumed by OSDs. **Not recommended** unless you have a very controlled environment where you will not risk formatting of devices with existing data. When `true`, all devices/partitions will be used. Is overridden by `deviceFilter` if specified.
- `deviceFilter`: A regular expression for short kernel names of devices (e.g. `sda`) that allows selection of devices to be consumed by OSDs. If individual devices have been specified for a node then this filter will be ignored. This field uses golang regular expression syntax. For example:
  - `sdb`: Only selects the `sdb` device if found
  - `^sd.`: Selects all devices starting with `sd`
  - `^sd[a-d]`: Selects devices starting with `sda`, `sdb`, `sdc`, and `sdd` if found
  - `^s`: Selects all devices that start with `s`
  - `^[^r]`: Selects all devices that do *not* start with `r`
- `devicePathFilter`: A regular expression for device paths (e.g. `/dev/disk/by-path/pci-0:1:2:3-scsi-1`) that allows selection of devices to be consumed by OSDs. If individual devices or `deviceFilter` have been specified for a node then this filter will be ignored. This field uses golang regular expression syntax. For example:

    - `^/dev/sd.`: Selects all devices starting with `sd`
    - `^/dev/disk/by-path/pci-.*`: Selects all devices which are connected to PCI bus
- `devices`: A list of individual device names belonging to this node to include in the storage cluster.
    - `name`: The name of the device (e.g., `sda`), or full udev path (e.g. `/dev/disk/by-id/ata-ST4000DM004-XXXX` - this will not change after reboots).
    - `config`: Device-specific config settings. See the config settings below
- `storageClassDeviceSets`: Explained in Storage Class Device Sets

Storage Class Device Sets

The following are the settings for Storage Class Device Sets which can be configured to create OSDs that are backed by block mode PVs.

- `name`: A name for the set.
- `count`: The number of devices in the set.
- `resources`: The CPU and RAM requests/limits for the devices. (Optional)
- `placement`: The placement criteria for the devices. (Optional) Default is no placement criteria.

  The syntax is the same as for other placement configuration. It supports `nodeAffinity`, `podAffinity`, `podAntiAffinity` and `tolerations` keys.

  It is recommended to configure the placement such that the OSDs will be as evenly spread across nodes as possible. At a minimum, anti-affinity should be added so at least one OSD will be placed on each available nodes.

  However, if there are more OSDs than nodes, this anti-affinity will not be effective. Another placement scheme to consider is to add labels to the nodes in such a way that the OSDs can be grouped on those nodes, create multiple storageClassDeviceSets, and add node affinity to each of the device sets that will place the OSDs in those sets of nodes.

- `portable`: If `true`, the OSDs will be allowed to move between nodes during failover. This requires a storage class that supports portability (e.g. `aws-ebs`, but not the local storage provisioner). If `false`, the OSDs will be assigned to a node permanently. Rook will configure Ceph's CRUSH map to support the portability.

- `tuneDeviceClass`: If `true`, because the OSD can be on a slow device class, Rook will adapt to that by tuning the OSD process. This will make Ceph perform better under that slow device.
- `volumeClaimTemplates`: A list of PVC templates to use for provisioning the underlying storage devices.
    - `resources.requests.storage`: The desired capacity for the underlying storage devices.
    - `storageClassName`: The StorageClass to provision PVCs from. Default would be to

use the cluster-default StorageClass. This StorageClass should provide a raw block device, multipath device, or logical volume. Other types are not supported.
    - `volumeMode`: The volume mode to be set for the PVC. Which should be Block
    - `accessModes`: The access mode for the PVC to be bound by OSD.
- `schedulerName`: Scheduler name for OSD pod placement. (Optional)
- `encrypted`: whether to encrypt all the OSDs in a given storageClassDeviceSet

OSD Configuration Settings

The following storage selection settings are specific to Ceph and do not apply to other backends. All variables are key-value pairs represented as strings.

- `metadataDevice`: Name of a device to use for the metadata of OSDs on each node. Performance can be improved by using a low latency device (such as SSD or NVMe) as the metadata device, while other spinning platter (HDD) devices on a node are used to store data. Provisioning will fail if the user specifies a `metadataDevice` but that device is not used as a metadata device by Ceph. Notably, `ceph-volume` will not use a device of the same device class (HDD, SSD, NVMe) as OSD devices for metadata, resulting in this failure.
- `storeType`: `bluestore`, the underlying storage format to use for each OSD. The default is set dynamically to `bluestore` for devices and is the only supported format at this point.
- `databaseSizeMB`: The size in MB of a bluestore database. Include quotes around the size.
- `walSizeMB`: The size in MB of a bluestore write ahead log (WAL). Include quotes around the size.
- `deviceClass`: The CRUSH device class to use for this selection of storage devices. (By default, if a device's class has not already been set, OSDs will automatically set a device's class to either `hdd`, `ssd`, or `nvme` based on the hardware properties exposed by the Linux kernel.) These storage classes can then be used to select the devices backing a storage pool by specifying them as the value of the pool spec's `deviceClass` field.
- `osdsPerDevice`**: The number of OSDs to create on each device. High performance devices such as NVMe can handle running multiple OSDs. If desired, this can be overridden for each node and each device.
- `encryptedDevice`**: Encrypt OSD volumes using dmcrypt ("true" or "false"). By default this option is disabled. See encryption for more information on encryption in Ceph.

** **NOTE**: Depending on the Ceph image running in your cluster, OSDs will be configured differently. Newer images will configure OSDs with `ceph-volume`, which provides support for `osdsPerDevice`, `encryptedDevice`, as well as other features that will be exposed in future Rook releases. OSDs created prior to Rook v0.9 or with older images of Luminous and Mimic are not created with `ceph-volume` and thus would not support the same features. For `ceph-volume`, the following images are supported:

- Luminous 12.2.10 or newer
- Mimic 13.2.3 or newer
- Nautilus

Storage Selection Via Ceph Drive Groups

Ceph Drive Groups allow for specifying highly advanced OSD layouts on nodes including non-homogeneous nodes. They are a way to describe a cluster layout using the properties of disks. It gives the user an abstract way tell Ceph which disks should turn into an OSD with which configuration without knowing the specifics of device names and paths. You can target specific disks by their device type, by vendor or model, by size, by whether they are rotational, and more. Disks with various properties can be specified to be data disks or wal/db disks.

As a brief example, let's assume nodes with 20 SSDs and 4 NVMe devices. A Drive Group could specify that all SSD devices should be data disks and two of the NVMe devices should be wal/db disks for the SSDs. That would leave two NVMe devices remaining for other usage, either for Ceph or another application.

Ceph supports adding devices as OSDs by Ceph Drive Group definitions in later versions of Ceph Octopus (v15.2.5+). See Ceph Drive Group docs for more info: https://docs.ceph.com/docs/master/cephadm/drivegroups/. Drive Groups cannot be used to provision OSDs on PVCs.

> **IMPORTANT:** When managing a Rook/Ceph cluster's OSD layouts with Drive Groups, the `storage` config is mostly ignored. `storageClassDeviceSets` can still be used to create OSDs on PVC, but Rook will no longer use `storage` configs for creating OSDs on a node's devices. To avoid confusion, we recommend using the `storage` config OR `driveGroups` and never both. Because `storage` and `driveGroups` should not be used simultaneously, Rook only supports provisioning OSDs with Drive Groups on new Rook-Ceph clusters.

A Drive Group is defined by a name, a Ceph Drive Group spec, and a Rook placement

- `name`: A name for the Drive Group.
- `spec`: The Ceph Drive group spec. Some components of the spec are treated differently in the context of Rook as noted below:
  - Rook overrides Ceph's definition of `placement` in order to use Rook's `placement` below.
  - Rook overrides Ceph's deprecated `host_pattern` in order to use Rook's `placement` below.
  - Rook overrides Ceph's `service_id` field to be the same as the Drive Group `name` above.
- `placement`: The placement criteria for nodes to provision with the Drive Group. (Optional) Default is no placement criteria, which matches all untainted nodes. The syntax is the same as for other placement configuration.

Annotations Configuration Settings

Annotations can be specified so that the Rook components will have those annotations

added to them.

You can set annotations for Rook components for the list of key value pairs:

- `all`: Set annotations for all components
- `mgr`: Set annotations for MGRs
- `mon`: Set annotations for mons
- `osd`: Set annotations for OSDs

When other keys are set, `all` will be merged together with the specific component.

Placement Configuration Settings

Placement configuration for the cluster services. It includes the following keys: `mgr`, `mon`, `osd`, `cleanup`, and `all`. Each service will have its placement configuration generated by merging the generic configuration under `all` with the most specific one (which will override any attributes).

NOTE: Placement of OSD pods is controlled using the Storage Class Device Set, not the general `placement` configuration.

A Placement configuration is specified (according to the kubernetes PodSpec) as:

- `nodeAffinity`: kubernetes NodeAffinity
- `podAffinity`: kubernetes PodAffinity
- `podAntiAffinity`: kubernetes PodAntiAffinity
- `tolerations`: list of kubernetes Toleration
- `topologySpreadConstraints`: kubernetes TopologySpreadConstraints

If you use `labelSelector` for `osd` pods, you must write two rules both for `rook-ceph-osd` and `rook-ceph-osd-prepare` like the example configuration. It comes from the design that there are these two pods for an OSD. For more detail, see the osd design doc and the related issue.

The Rook Ceph operator creates a Job called `rook-ceph-detect-version` to detect the full Ceph version used by the given `cephVersion.image`. The placement from the `mon` section is used for the Job except for the `PodAntiAffinity` field.

Cluster-wide Resources Configuration Settings

Resources should be specified so that the Rook components are handled after Kubernetes Pod Quality of Service classes. This allows to keep Rook components running when for example a node runs out of memory and the Rook components are not killed depending on their Quality of Service class.

You can set resource requests/limits for Rook components through the Resource Requirements/Limits structure in the following keys:

- `mgr`: Set resource requests/limits for MGRs

- `mon`: Set resource requests/limits for mons
- `osd`: Set resource requests/limits for OSDs
- `prepareosd`: Set resource requests/limits for OSD prepare job
- `crashcollector`: Set resource requests/limits for crash. This pod runs wherever there is a Ceph pod running. It scrapes for Ceph daemon core dumps and sends them to the Ceph manager crash module so that core dumps are centralized and can be easily listed/accessed. You can read more about the Ceph Crash module.
- `cleanup`: Set resource requests/limits for cleanup job, responsible for wiping cluster's data after uninstall

In order to provide the best possible experience running Ceph in containers, Rook internally enforces minimum memory limits if resource limits are passed. If a user configures a limit or request value that is too low, Rook will refuse to run the pod(s). Here are the current minimum amounts of memory in MB to apply so that Rook will agree to run Ceph pods:

- `mon`: 1024MB
- `mgr`: 512MB
- `osd`: 2048MB
- `mds`: 4096MB

Rook does not enforce any minimum limit nor request on the following:

- prepare OSD pod: This pod commonly takes up to 50MB, but depending on the OSD scenario may need more memory. 100MB would be more conservative.
- crashcollector pod: This pod commonly takes around 60MB.

Resource Requirements/Limits

For more information on resource requests/limits see the official Kubernetes documentation: Kubernetes - Managing Compute Resources for Containers

- `requests`: Requests for cpu or memory.
  - `cpu`: Request for CPU (example: one CPU core `1`, 50% of one CPU core `500m`).
  - `memory`: Limit for Memory (example: one gigabyte of memory `1Gi`, half a gigabyte of memory `512Mi`).
- `limits`: Limits for cpu or memory.
  - `cpu`: Limit for CPU (example: one CPU core `1`, 50% of one CPU core `500m`).
  - `memory`: Limit for Memory (example: one gigabyte of memory `1Gi`, half a gigabyte of memory `512Mi`).

Priority Class Names Configuration Settings

Priority class names can be specified so that the Rook components will have those priority class names added to them.

You can set priority class names for Rook components for the list of key value pairs:

- `all`: Set priority class names for MGRs, Mons, OSDs.
- `mgr`: Set priority class names for MGRs.
- `mon`: Set priority class names for Mons.
- `osd`: Set priority class names for OSDs.

The specific component keys will act as overrides to `all`.

Health settings

Rook-Ceph will monitor the state of the CephCluster on various components by default. The following CRD settings are available:

- `healthCheck`: main ceph cluster health monitoring section

Currently three health checks are implemented:

- `mon`: health check on the ceph monitors, basically check whether monitors are members of the quorum. If after a certain timeout a given monitor has not joined the quorum back it will be failed over and replace by a new monitor.
- `osd`: health check on the ceph osds
- `status`: ceph health status check, periodically check the Ceph health state and reflects it in the CephCluster CR status field.

The liveness probe of each daemon can also be controlled via `livenessProbe`, the setting is valid for `mon`, `mgr` and `osd`. Here is a complete example for both `daemonHealth` and `livenessProbe`:

```
healthCheck:
  daemonHealth:
    mon:
      disabled: false
      interval: 45s
      timeout: 600s
    osd:
      disabled: false
      interval: 60s
    status:
      disabled: false
  livenessProbe:
    mon:
      disabled: false
    mgr:
      disabled: false
    osd:
      disabled: false
```

The probe itself can also be overridden, refer to the Kubernetes documentation.

For example, you could change the `mgr` probe by applying:

```
healthCheck:
  livenessProbe:
    mgr:
      disabled: false
      probe:
        httpGet:
          path: /
          port: 9283
        initialDelaySeconds: 3
        periodSeconds: 3
```

Changing the liveness probe is an advanced operation and should rarely be necessary. If you want to change these settings, start with the probe spec Rook generates by default and then modify the desired settings.

Samples

Here are several samples for configuring Ceph clusters. Each of the samples must also include the namespace and corresponding access granted for management by the Ceph operator. See the common cluster resources below.

Storage configuration: All devices

```
apiVersion: ceph.rook.io/v1
kind: CephCluster
metadata:
  name: rook-ceph
  namespace: rook-ceph
spec:
  cephVersion:
    image: ceph/ceph:v15.2.4
  dataDirHostPath: /var/lib/rook
  mon:
    count: 3
    allowMultiplePerNode: true
  dashboard:
    enabled: true
  # cluster level storage configuration and selection
  storage:
    useAllNodes: true
    useAllDevices: true
    deviceFilter:
    config:
      metadataDevice:
      databaseSizeMB: "1024" # this value can be removed for environments with normal sized disks (100 GB or larger)
      journalSizeMB: "1024"  # this value can be removed for environments with normal sized disks (20 GB or larger)
      osdsPerDevice: "1"
```

Storage Configuration: Specific devices

Individual nodes and their config can be specified so that only the named nodes below will be used as storage resources. Each node's 'name' field should match their 'kubernetes.io/hostname' label.

```
apiVersion: ceph.rook.io/v1
kind: CephCluster
metadata:
  name: rook-ceph
  namespace: rook-ceph
spec:
  cephVersion:
    image: ceph/ceph:v15.2.4
  dataDirHostPath: /var/lib/rook
  mon:
    count: 3
    allowMultiplePerNode: true
  dashboard:
    enabled: true
  # cluster level storage configuration and selection
  storage:
    useAllNodes: false
    useAllDevices: false
    deviceFilter:
    config:
      metadataDevice:
      databaseSizeMB: "1024" # this value can be removed for environments with normal sized disks (100 GB or
larger)
    nodes:
    - name: "172.17.4.201"
      devices:             # specific devices to use for storage can be specified for each node
      - name: "sdb" # Whole storage device
      - name: "sdc1" # One specific partition. Should not have a file system on it.
      - name: "/dev/disk/by-id/ata-ST4000DM004-XXXX" # both device name and explicit udev links are supported
      config:         # configuration can be specified at the node level which overrides the cluster level
config
        storeType: bluestore
    - name: "172.17.4.301"
      deviceFilter: "^sd."
```

Node Affinity

To control where various services will be scheduled by kubernetes, use the placement configuration sections below. The example under 'all' would have all services scheduled on kubernetes nodes labeled with 'role=storage-node' and tolerate taints with a key of 'storage-node'.

```
apiVersion: ceph.rook.io/v1
kind: CephCluster
metadata:
```

```
  name: rook-ceph
  namespace: rook-ceph
spec:
  cephVersion:
    image: ceph/ceph:v15.2.4
  dataDirHostPath: /var/lib/rook
  mon:
    count: 3
    allowMultiplePerNode: true
  # enable the ceph dashboard for viewing cluster status
  dashboard:
    enabled: true
  placement:
    all:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: role
              operator: In
              values:
              - storage-node
      tolerations:
      - key: storage-node
        operator: Exists
    mgr:
      nodeAffinity:
      tolerations:
    mon:
      nodeAffinity:
      tolerations:
    osd:
      nodeAffinity:
      tolerations:
```

Resource Requests/Limits

To control how many resources the Rook components can request/use, you can set requests and limits in Kubernetes for them. You can override these requests/limits for OSDs per node when using `useAllNodes: false` in the `node` item in the `nodes` list.

> **WARNING**: Before setting resource requests/limits, please take a look at the Ceph documentation for recommendations for each component: Ceph - Hardware Recommendations.

```
apiVersion: ceph.rook.io/v1
kind: CephCluster
metadata:
  name: rook-ceph
  namespace: rook-ceph
spec:
  cephVersion:
```

```
    image: ceph/ceph:v15.2.4
  dataDirHostPath: /var/lib/rook
  mon:
    count: 3
    allowMultiplePerNode: true
  # enable the ceph dashboard for viewing cluster status
  dashboard:
    enabled: true
  # cluster level resource requests/limits configuration
  resources:
  storage:
    useAllNodes: false
    nodes:
    - name: "172.17.4.201"
      resources:
        limits:
          cpu: "2"
          memory: "4096Mi"
        requests:
          cpu: "2"
          memory: "4096Mi"
```

OSD Topology

The topology of the cluster is important in production environments where you want your data spread across failure domains. The topology can be controlled by adding labels to the nodes. When the labels are found on a node at first OSD deployment, Rook will add them to the desired level in the CRUSH map.

The complete list of labels in hierarchy order from highest to lowest is:

```
topology.kubernetes.io/region
topology.kubernetes.io/zone
topology.rook.io/datacenter
topology.rook.io/room
topology.rook.io/pod
topology.rook.io/pdu
topology.rook.io/row
topology.rook.io/rack
topology.rook.io/chassis
```

For example, if the following labels were added to a node:

```
kubectl label node mynode topology.kubernetes.io/zone=zone1
kubectl label node mynode topology.rook.io/rack=rack1
```

> For versions previous to K8s 1.17, use the topology key: failure-domain.beta.kubernetes.io/zone or region

These labels would result in the following hierarchy for OSDs on that node (this

command can be run in the Rook toolbox):

```
[root@mynode /]# ceph osd tree
ID CLASS WEIGHT  TYPE NAME                STATUS REWEIGHT PRI-AFF
-1       0.01358 root default
-5       0.01358     zone zone1
-4       0.01358         rack rack1
-3       0.01358             host mynode
 0   hdd 0.00679                 osd.0         up  1.00000 1.00000
 1   hdd 0.00679                 osd.1         up  1.00000 1.00000
```

Ceph requires unique names at every level in the hierarchy (CRUSH map). For example, you cannot have two racks with the same name that are in different zones. Racks in different zones must be named uniquely.

Note that the `host` is added automatically to the hierarchy by Rook. The host cannot be specified with a topology label. All topology labels are optional.

> **HINT** When setting the node labels prior to `CephCluster` creation, these settings take immediate effect. However, applying this to an already deployed `CephCluster` requires removing each node from the cluster first and then re-adding it with new configuration to take effect. Do this node by node to keep your data safe! Check the result with `ceph osd tree` from the Rook Toolbox. The OSD tree should display the hierarchy for the nodes that already have been re-added.

To utilize the `failureDomain` based on the node labels, specify the corresponding option in the CephBlockPool

```
apiVersion: ceph.rook.io/v1
kind: CephBlockPool
metadata:
  name: replicapool
  namespace: rook-ceph
spec:
  failureDomain: rack  # this matches the topology labels on nodes
  replicated:
    size: 3
```

This configuration will split the replication of volumes across unique racks in the data center setup.

Using PVC storage for monitors

In the CRD specification below three monitors are created each using a 10Gi PVC created by Rook using the `local-storage` storage class.

```
apiVersion: ceph.rook.io/v1
kind: CephCluster
metadata:
  name: rook-ceph
  namespace: rook-ceph
```

```
spec:
  cephVersion:
    image: ceph/ceph:v15.2.4
  dataDirHostPath: /var/lib/rook
  mon:
    count: 3
    allowMultiplePerNode: false
    volumeClaimTemplate:
      spec:
        storageClassName: local-storage
        resources:
          requests:
            storage: 10Gi
  dashboard:
    enabled: true
  storage:
    useAllNodes: true
    useAllDevices: true
    deviceFilter:
    config:
      metadataDevice:
      databaseSizeMB: "1024" # this value can be removed for environments with normal sized disks (100 GB or larger)
      journalSizeMB: "1024"  # this value can be removed for environments with normal sized disks (20 GB or larger)
      osdsPerDevice: "1"
```

Using StorageClassDeviceSets

In the CRD specification below, 3 OSDs (having specific placement and resource values) and 3 mons with each using a 10Gi PVC, are created by Rook using the `local-storage` storage class.

```
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: local-storage
        resources:
          requests:
            storage: 10Gi
  cephVersion:
    image: ceph/ceph:v15.2.4
    allowUnsupported: false
```

```
  dashboard:
    enabled: true
  network:
    hostNetwork: false
  storage:
    storageClassDeviceSets:
    - name: set1
      count: 3
      portable: false
      tuneDeviceClass: false
      resources:
        limits:
          cpu: "500m"
          memory: "4Gi"
        requests:
          cpu: "500m"
          memory: "4Gi"
      placement:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 100
            podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: "rook.io/cluster"
                  operator: In
                  values:
                    - cluster1
                topologyKey: "topology.kubernetes.io/zone"
      volumeClaimTemplates:
      - metadata:
          name: data
        spec:
          resources:
            requests:
              storage: 10Gi
          storageClassName: local-storage
          volumeMode: Block
          accessModes:
            - ReadWriteOnce
```

Dedicated metadata and wal device for OSD on PVC

In the simplest case, Ceph OSD BlueStore consumes a single (primary) storage device. BlueStore is the engine used by the OSD to store data.

The storage device is normally used as a whole, occupying the full device that is managed directly by BlueStore. It is also possible to deploy BlueStore across additional devices such as a DB device. This device can be used for storing BlueStore’s internal metadata. BlueStore (or rather, the embedded RocksDB) will put as much metadata as it can on the DB device to improve performance. If the DB device

fills up, metadata will spill back onto the primary device (where it would have been otherwise). Again, it is only helpful to provision a DB device if it is faster than the primary device.

You can have multiple `volumeClaimTemplates` where each might either represent a device or a metadata device. So just taking the `storage` section this will give something like:

```
  storage:
   storageClassDeviceSets:
    - name: set1
      count: 3
      portable: false
      tuneDeviceClass: false
      volumeClaimTemplates:
      - metadata:
          name: data
        spec:
          resources:
            requests:
              storage: 10Gi
          # IMPORTANT: Change the storage class depending on your environment (e.g. local-storage, gp2)
          storageClassName: gp2
          volumeMode: Block
          accessModes:
            - ReadWriteOnce
      - metadata:
          name: metadata
        spec:
          resources:
            requests:
              # Find the right size https://docs.ceph.com/docs/master/rados/configuration/bluestore-config-
ref/#sizing
              storage: 5Gi
          # IMPORTANT: Change the storage class depending on your environment (e.g. local-storage, io1)
          storageClassName: io1
          volumeMode: Block
          accessModes:
            - ReadWriteOnce
```

> **NOTE**: Note that Rook only supports three naming convention for a given template:

- “data”: represents the main OSD block device, where your data is being stored.
- “metadata”: represents the metadata (including block.db and block.wal) device used to store the Ceph Bluestore database for an OSD.
- “wal”: represents the block.wal device used to store the Ceph Bluestore database for an OSD. If this device is set, “metadata” device will refer specifically to block.db device. It is recommended to use a faster storage class for the metadata or wal device, with a slower device for the data. Otherwise, having a separate metadata device will not improve the performance.

The bluestore partition has the following reference combinations supported by the

ceph-volume utility:

- A single “data” device. ```yaml storage: storageClassDeviceSets:
    - name: set1 count: 3 portable: false tuneDeviceClass: false volumeClaimTemplates:
        - metadata: name: data spec: resources: requests: storage: 10Gi # IMPORTANT: Change the storage class depending on your environment (e.g. local-storage, gp2) storageClassName: gp2 volumeMode: Block accessModes: - ReadWriteOnce ```
- A “data” device and a “metadata” device. ```yaml storage: storageClassDeviceSets:
    - name: set1 count: 3 portable: false tuneDeviceClass: false volumeClaimTemplates:
        - metadata: name: data spec: resources: requests: storage: 10Gi # IMPORTANT: Change the storage class depending on your environment (e.g. local-storage, gp2) storageClassName: gp2 volumeMode: Block accessModes: - ReadWriteOnce
        - metadata: name: metadata spec: resources: requests: # Find the right size https://docs.ceph.com/docs/master/rados/configuration/bluestore-config-ref/#sizing storage: 5Gi # IMPORTANT: Change the storage class depending on your environment (e.g. local-storage, io1) storageClassName: io1 volumeMode: Block accessModes: - ReadWriteOnce ```
- A “data” device and a “wal” device. A WAL device can be used for BlueStore’s internal journal or write-ahead log (block.wal), it is only useful to use a WAL device if the device is faster than the primary device (data device). There is no separate “metadata” device in this case, the data of main OSD block and block.db located in “data” device. ```yaml storage: storageClassDeviceSets:
    - name: set1 count: 3 portable: false tuneDeviceClass: false volumeClaimTemplates:
        - metadata: name: data spec: resources: requests: storage: 10Gi # IMPORTANT: Change the storage class depending on your environment (e.g. local-storage, gp2) storageClassName: gp2 volumeMode: Block accessModes: - ReadWriteOnce
        - metadata: name: wal spec: resources: requests: # Find the right size https://docs.ceph.com/docs/master/rados/configuration/bluestore-config-ref/#sizing storage: 5Gi # IMPORTANT: Change the storage class depending on your environment (e.g. local-storage, io1) storageClassName: io1 volumeMode: Block accessModes: - ReadWriteOnce ```
- A “data” device, a “metadata” device and a “wal” device. ```yaml storage: storageClassDeviceSets:
    - name: set1 count: 3 portable: false tuneDeviceClass: false volumeClaimTemplates:
        - metadata: name: data spec: resources: requests: storage: 10Gi # IMPORTANT: Change the storage class depending on your environment (e.g. local-storage, gp2) storageClassName: gp2 volumeMode: Block accessModes: - ReadWriteOnce
        - metadata: name: metadata spec: resources: requests: # Find the right

size https://docs.ceph.com/docs/master/rados/configuration/bluestore-config-ref/#sizing storage: 5Gi # IMPORTANT: Change the storage class depending on your environment (e.g. local-storage, io1) storageClassName: io1 volumeMode: Block accessModes: - ReadWriteOnce
- metadata: name: wal spec: resources: requests: # Find the right size https://docs.ceph.com/docs/master/rados/configuration/bluestore-config-ref/#sizing storage: 5Gi # IMPORTANT: Change the storage class depending on your environment (e.g. local-storage, io1) storageClassName: io1 volumeMode: Block accessModes: - ReadWriteOnce ```

To determine the size of the metadata block follow the official Ceph sizing guide.

With the present configuration, each OSD will have its main block allocated a 10GB device as well a 5GB device to act as a bluestore database.

External cluster

The minimum supported Ceph version for the External Cluster is Luminous 12.2.x.

The features available from the external cluster will vary depending on the version of Ceph. The following table shows the minimum version of Ceph for some of the features:

FEATURE	CEPH VERSION
Dynamic provisioning RBD	12.2.X
Configure extra CRDs (object, file, nfs)[1]	13.2.3
Dynamic provisioning CephFS	14.2.3

Pre-requisites

In order to configure an external Ceph cluster with Rook, we need to inject some information in order to connect to that cluster. You can use the `cluster/examples/kubernetes/ceph/import-external-cluster.sh` script to achieve that. The script will look for the following populated environment variables:

- `NAMESPACE`: the namespace where the configmap and secrets should be injected
- `ROOK_EXTERNAL_FSID`: the fsid of the external Ceph cluster, it can be retrieved via the `ceph fsid` command
- `ROOK_EXTERNAL_CEPH_MON_DATA`: is a common-separated list of running monitors IP address along with their ports, e.g: `a=172.17.0.4:6789,b=172.17.0.5:6789,c=172.17.0.6:6789`. You don't need to specify all the monitors, you can simply pass one and the Operator will discover the rest. The name of the monitor is the name that appears in the `ceph status` output.

Now, we need to give Rook a key to connect to the cluster in order to perform various operations such as health cluster check, CSI keys management etc… It is recommended to generate keys with minimal access so the admin key does not need to be used by the external cluster. In this case, the admin key is only needed to generate the keys that

will be used by the external cluster. But if the admin key is to be used by the external cluster, set the following variable:

- `ROOK_EXTERNAL_ADMIN_SECRET`: **OPTIONAL:** the external Ceph cluster admin secret key, it can be retrieved via the `ceph auth get-key client.admin` command.

> **WARNING**: If you plan to create CRs (pool, rgw, mds, nfs) in the external cluster, you **MUST** inject the client.admin keyring as well as injecting `cluster-external-management.yaml`

Example:

```
export NAMESPACE=rook-ceph-external
export ROOK_EXTERNAL_FSID=3240b4aa-ddbc-42ee-98ba-4ea7b2a61514
export ROOK_EXTERNAL_CEPH_MON_DATA=a=172.17.0.4:6789
export ROOK_EXTERNAL_ADMIN_SECRET=AQC6Ylxdja+NDBAAB7qy9MEAr4VLLq4dCIvxtg==
```

If the Ceph admin key is not provided, the following script needs to be executed on a machine that can connect to the Ceph cluster using the Ceph admin key. On that machine, run `cluster/examples/kubernetes/ceph/create-external-cluster-resources.sh`. The script will automatically create users and keys with the lowest possible privileges and populate the necessary environment variables for `cluster/examples/kubernetes/ceph/import-external-cluster.sh` to work correctly.

Finally, you can simply execute the script like this from a machine that has access to your Kubernetes cluster:

```
bash cluster/examples/kubernetes/ceph/import-external-cluster.sh
```

CephCluster example (consumer)

Assuming the above section has successfully completed, here is a CR example:

```
apiVersion: ceph.rook.io/v1
kind: CephCluster
metadata:
  name: rook-ceph-external
  namespace: rook-ceph-external
spec:
  external:
    enable: true
  crashCollector:
    disable: true
  # optionally, the ceph-mgr IP address can be pass to gather metric from the prometheus exporter
  #monitoring:
    #enabled: true
    #rulesNamespace: rook-ceph
    #externalMgrEndpoints:
      #- ip: 192.168.39.182
```

Choose the namespace carefully, if you have an existing cluster managed by Rook, you have likely already injected `common.yaml` . Additionally, you now need to inject `common-external.yaml` too.

You can now create it like this:

```
kubectl create -f cluster/examples/kubernetes/ceph/cluster-external.yaml
```

If the previous section has not been completed, the Rook Operator will still acknowledge the CR creation but will wait forever to receive connection information.

> **WARNING**: If no cluster is managed by the current Rook Operator, you need to inject `common.yaml` , then modify `cluster-external.yaml` and specify `rook-ceph` as `namespace` .

If this is successfull you will see the CepCluster status as connected.

```
kubectl get CephCluster -n rook-ceph-external
NAME                 DATADIRHOSTPATH   MONCOUNT   AGE    STATE       HEALTH
rook-ceph-external   /var/lib/rook               162m   Connected   HEALTH_OK
```

Before you create a StorageClass with this cluster you will need to create a Pool in your external Ceph Cluster.

Example StorageClass based on external Ceph Pool

In Ceph Cluster let us list the pools available:

```
rados df
POOL_NAME     USED OBJECTS CLONES COPIES MISSING_ON_PRIMARY UNFOUND DEGRADED RD_OPS  RD WR_OPS  WR USED COMPR
UNDER COMPR
replicated_2g  0 B       0      0      0                  0       0        0      0 0 B      0 0 B        0 B
0 B
```

Here is an example StorageClass configuration that uses the `replicated_2g` pool from the external cluster:

```
cat << EOF | kubectl apply -f -
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: rook-ceph-block-ext
# Change "rook-ceph" provisioner prefix to match the operator namespace if needed
provisioner: rook-ceph.rbd.csi.ceph.com
parameters:
    # clusterID is the namespace where the rook cluster is running
    clusterID: rook-ceph-external
    # Ceph pool into which the RBD image shall be created
    pool: replicated_2g

    # RBD image format. Defaults to "2".
```

```
    imageFormat: "2"

    # RBD image features. Available for imageFormat: "2". CSI RBD currently supports only `layering` feature.
    imageFeatures: layering

    # The secrets contain Ceph admin credentials.
    csi.storage.k8s.io/provisioner-secret-name: rook-csi-rbd-provisioner
    csi.storage.k8s.io/provisioner-secret-namespace: rook-ceph-external
    csi.storage.k8s.io/controller-expand-secret-name: rook-csi-rbd-provisioner
    csi.storage.k8s.io/controller-expand-secret-namespace: rook-ceph-external
    csi.storage.k8s.io/node-stage-secret-name: rook-csi-rbd-node
    csi.storage.k8s.io/node-stage-secret-namespace: rook-ceph-external

    # Specify the filesystem type of the volume. If not specified, csi-provisioner
    # will set default as `ext4`. Note that `xfs` is not recommended due to potential deadlock
    # in hyperconverged settings where the volume is mounted on the same node as the osds.
    csi.storage.k8s.io/fstype: ext4

# Delete the rbd volume when a PVC is deleted
reclaimPolicy: Delete
allowVolumeExpansion: true
EOF
```

You can now create a persistent volume based on this StorageClass.

CephCluster example (management)

The following CephCluster CR represents a cluster that will perform management tasks on the external cluster. It will not only act as a consumer but will also allow the deployment of other CRDs such as CephFilesystem or CephObjectStore. As mentioned above, you would need to inject the admin keyring for that.

The corresponding YAML example:

```
apiVersion: ceph.rook.io/v1
kind: CephCluster
metadata:
  name: rook-ceph-external
  namespace: rook-ceph-external
spec:
  external:
    enable: true
  dataDirHostPath: /var/lib/rook
  cephVersion:
    image: ceph/ceph:v15.2.4 # Should match external cluster version
```

Cleanup policy

Rook has the ability to cleanup resources and data that were deployed when a `delete cephcluster` command is issued. The policy represents the confirmation that cluster data

should be forcibly deleted. The cleanupPolicy should only be added to the cluster when the cluster is about to be deleted. After the `confirmation` field of the cleanup policy is set, Rook will stop configuring the cluster as if the cluster is about to be destroyed in order to prevent these settings from being deployed unintentionally. The `cleanupPolicy` CR settings has different fields:

- `confirmation`: Only an empty string and `yes-really-destroy-data` are valid values for this field. If an empty string is set, Rook will only remove Ceph's metadata. A re-installation will not be possible unless the hosts are cleaned first. If `yes-really-destroy-data` the operator will automatically delete data on the hostpath of cluster nodes and clean devices with OSDs. The cluster can then be re-installed if desired with no further steps.
- `sanitizeDisks`: sanitizeDisks represents advanced settings that can be used to sanitize drives. This field only affects if `confirmation` is set to `yes-really-destroy-data`. However, the administrator might want to sanitize the drives in more depth with the following flags:
  - `method`: indicates if the entire disk should be sanitized or simply ceph's metadata. Possible choices are 'quick' (default) or 'complete'
  - `dataSource`: indicate where to get random bytes from to write on the disk. Possible choices are 'zero' (default) or 'random'. Using random sources will consume entropy from the system and will take much more time then the zero source
  - `iteration`: overwrite N times instead of the default (1). Takes an integer value

To automate activation of the cleanup, you can use the following command. **WARNING: DATA WILL BE PERMANENTLY DELETED**:

```
kubectl -n rook-ceph patch cephcluster rook-ceph --type merge -p '{"spec":{"cleanupPolicy":
{"confirmation":"yes-really-destroy-data"}}}'
```

Nothing will happen until the deletion of the CR is requested, so this can still be reverted. However, all new configuration by the operator will be blocked with this cleanup policy enabled.

1. Configure an object store, shared filesystem, or NFS resources in the local cluster to connect to the external Ceph cluster ↩

Ceph Block Pool CRD

Rook allows creation and customization of storage pools through the custom resource definitions (CRDs). The following settings are available for pools.

Samples

Replicated

For optimal performance, while also adding redundancy, this sample will configure Ceph to make three full copies of the data on multiple nodes.

> **NOTE**: This sample requires *at least 1 OSD per node*, with each OSD located on *3 different nodes*.

Each OSD must be located on a different node, because the `failureDomain` is set to `host` and the `replicated.size` is set to `3`.

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

Erasure Coded

This sample will lower the overall storage capacity requirement, while also adding redundancy by using erasure coding.

> **NOTE**: This sample requires *at least 3 bluestore OSDs*.

The OSDs can be located on a single Ceph node or spread across multiple nodes, because the `failureDomain` is set to `osd` and the `erasureCoded` chunk settings require at least 3 different OSDs (2 `dataChunks` + 1 `codingChunks`).

```
apiVersion: ceph.rook.io/v1
kind: CephBlockPool
metadata:
  name: ecpool
  namespace: rook-ceph
spec:
  failureDomain: osd
```

```
  erasureCoded:
    dataChunks: 2
    codingChunks: 1
  deviceClass: hdd
```

High performance applications typically will not use erasure coding due to the performance overhead of creating and distributing the chunks in the cluster.

When creating an erasure-coded pool, it is highly recommended to create the pool when you have **bluestore OSDs** in your cluster (see the OSD configuration settings. Filestore OSDs have limitations that are unsafe and lower performance.

Pool Settings

Metadata

- `name`: The name of the pool to create.
- `namespace`: The namespace of the Rook cluster where the pool is created.

Spec

- `replicated`: Settings for a replicated pool. If specified, `erasureCoded` settings must not be specified.
  - `size`: The desired number of copies to make of the data in the pool.
  - `requireSafeReplicaSize`: set to false if you want to create a pool with size 1, setting pool size 1 could lead to data loss without recovery. Make sure you are *ABSOLUTELY CERTAIN* that is what you want.
- `erasureCoded`: Settings for an erasure-coded pool. If specified, `replicated` settings must not be specified. See below for more details on erasure coding.
  - `dataChunks`: Number of chunks to divide the original object into
  - `codingChunks`: Number of coding chunks to generate
- `failureDomain`: The failure domain across which the data will be spread. This can be set to a value of either `osd` or `host`, with `host` being the default setting. A failure domain can also be set to a different type (e.g. `rack`), if it is added as a `location` in the Storage Selection Settings. If a `replicated` pool of size `3` is configured and the `failureDomain` is set to `host`, all three copies of the replicated data will be placed on OSDs located on `3` different Ceph hosts. This case is guaranteed to tolerate a failure of two hosts without a loss of data. Similarly, a failure domain set to `osd`, can tolerate a loss of two OSD devices.

  If erasure coding is used, the data and coding chunks are spread across the configured failure domain.

  > **NOTE**: Neither Rook, nor Ceph, prevent the creation of a cluster where the replicated data (or Erasure Coded chunks) can be written safely. By design, Ceph will delay checking for suitable OSDs until a write request

> is made and this write can hang if there are not sufficient OSDs to satisfy the request.

- `deviceClass` : Sets up the CRUSH rule for the pool to distribute data only on the specified device class. If left empty or unspecified, the pool will use the cluster's default CRUSH root, which usually distributes data over all OSDs, regardless of their class.
- `crushRoot` : The root in the crush map to be used by the pool. If left empty or unspecified, the default root will be used. Creating a crush hierarchy for the OSDs currently requires the Rook toolbox to run the Ceph tools described here.
- `enableRBDStats` : Enables collecting RBD per-image IO statistics by enabling dynamic OSD performance counters. Defaults to false. For more info see the ceph documentation.
- `parameters` : Sets any parameters listed to the given pool
  - `target_size_ratio:` gives a hint (%) to Ceph in terms of expected consumption of the total cluster capacity of a given pool, for more info see the ceph documentation
  - `compression_mode` : Sets up the pool for inline compression when using a Bluestore OSD. If left unspecified does not setup any compression mode for the pool. Values supported are the same as Bluestore inline compression modes, such as `none` , `passive` , `aggressive` , and `force` .

Add specific pool properties

With `poolProperties` you can set any pool property:

```
spec:
  parameters:
    <name of the parameter>: <parameter value>
```

For instance:

```
spec:
  parameters:
    min_size: 1
```

Erasure Coding

Erasure coding allows you to keep your data safe while reducing the storage overhead. Instead of creating multiple replicas of the data, erasure coding divides the original data into chunks of equal size, then generates extra chunks of that same size for redundancy.

For example, if you have an object of size 2MB, the simplest erasure coding with two

data chunks would divide the object into two chunks of size 1MB each (data chunks). One more chunk (coding chunk) of size 1MB will be generated. In total, 3MB will be stored in the cluster. The object will be able to suffer the loss of any one of the chunks and still be able to reconstruct the original object.

The number of data and coding chunks you choose will depend on your resiliency to loss and how much storage overhead is acceptable in your storage cluster. Here are some examples to illustrate how the number of chunks affects the storage and loss toleration.

Data chunks (k)	Coding chunks (m)	Total storage	Losses Tolerated	OSDs required
2	1	1.5x	1	3
2	2	2x	2	4
4	2	1.5x	2	6
16	4	1.25x	4	20

The `failureDomain` must be also be taken into account when determining the number of chunks. The failure domain determines the level in the Ceph CRUSH hierarchy where the chunks must be uniquely distributed. This decision will impact whether node losses or disk losses are tolerated. There could also be performance differences of placing the data across nodes or osds.

- `host`: All chunks will be placed on unique hosts
- `osd`: All chunks will be placed on unique OSDs

If you do not have a sufficient number of hosts or OSDs for unique placement the pool can be created, writing to the pool will hang.

Rook currently only configures two levels in the CRUSH map. It is also possible to configure other levels such as `rack` with by adding topology labels to the nodes.

Ceph Object Store CRD

Rook allows creation and customization of object stores through the custom resource definitions (CRDs). The following settings are available for Ceph object stores.

Sample

Erasure Coded

Erasure coded pools require the OSDs to use `bluestore` for the configured `storeType`. Additionally, erasure coded pools can only be used with `dataPools`. The `metadataPool` must use a replicated pool.

> **NOTE**: This sample requires *at least 3 bluestore OSDs*, with each OSD located on a *different node*.

The OSDs must be located on different nodes, because the `failureDomain` is set to `host` and the `erasureCoded` chunk settings require at least 3 different OSDs (2 `dataChunks` + 1 `codingChunks`).

```
apiVersion: ceph.rook.io/v1
kind: CephObjectStore
metadata:
  name: my-store
  namespace: rook-ceph
spec:
  metadataPool:
    failureDomain: host
    replicated:
      size: 3
  dataPool:
    failureDomain: host
    erasureCoded:
      dataChunks: 2
      codingChunks: 1
  preservePoolsOnDelete: true
  gateway:
    type: s3
    sslCertificateRef:
    port: 80
    securePort:
    instances: 1
    # A key/value list of annotations
    annotations:
    #  key: value
    placement:
    #  nodeAffinity:
    #    requiredDuringSchedulingIgnoredDuringExecution:
    #      nodeSelectorTerms:
```

```
    #      - matchExpressions:
    #        - key: role
    #          operator: In
    #          values:
    #          - rgw-node
    #  tolerations:
    #  - key: rgw-node
    #    operator: Exists
    #  podAffinity:
    #  podAntiAffinity:
    #  topologySpreadConstraints:
    resources:
    #  limits:
    #    cpu: "500m"
    #    memory: "1024Mi"
    #  requests:
    #    cpu: "500m"
    #    memory: "1024Mi"
  #zone:
    #name: zone-a
```

Object Store Settings

Metadata

- `name`: The name of the object store to create, which will be reflected in the pool and other resource names.
- `namespace`: The namespace of the Rook cluster where the object store is created.

Pools

The pools allow all of the settings defined in the Pool CRD spec. For more details, see the Pool CRD settings. In the example above, there must be at least three hosts (size 3) and at least three devices (2 data + 1 coding chunks) in the cluster.

When the `zone` section is set pools with the object stores name will not be created since the object-store will the using the pools created by the ceph-object-zone.

- `metadataPool`: The settings used to create all of the object store metadata pools. Must use replication.
- `dataPool`: The settings to create the object store data pool. Can use replication or erasure coding.
- `preservePoolsOnDelete`: If it is set to ‘true’ the pools used to support the object store will remain when the object store will be deleted. This is a security measure to avoid accidental loss of data. It is set to ‘false’ by default. If not specified is also deemed as ‘false’.

Gateway Settings

The gateway settings correspond to the RGW daemon settings.

- `type`: `S3` is supported
- `sslCertificateRef`: If the certificate is not specified, SSL will not be configured. If specified, this is the name of the Kubernetes secret that contains the SSL certificate to be used for secure connections to the object store. Rook will look in the secret provided at the `cert` key name. The value of the `cert` key must be in the format expected by the RGW service: "The server key, server certificate, and any other CA or intermediate certificates be supplied in one file. Each of these items must be in pem form."
- `port`: The port on which the Object service will be reachable. If host networking is enabled, the RGW daemons will also listen on that port. If running on SDN, the RGW daemon listening port will be 8080 internally.
- `securePort`: The secure port on which RGW pods will be listening. An SSL certificate must be specified.
- `instances`: The number of pods that will be started to load balance this object store.
- `externalRgwEndpoints`: A list of IP addresses to connect to external existing Rados Gateways (works with external mode). This setting will be ignored if the `CephCluster` does not have `external` spec enabled. Refer to the external cluster section for more details.
- `annotations`: Key value pair list of annotations to add.
- `placement`: The Kubernetes placement settings to determine where the RGW pods should be started in the cluster.
- `resources`: Set resource requests/limits for the Gateway Pod(s), see Resource Requirements/Limits.
- `priorityClassName`: Set priority class name for the Gateway Pod(s)

Example of external rgw endpoints to connect to:

```
gateway:
  port: 80
  externalRgwEndpoints:
    - ip: 192.168.39.182
```

This will create a service with the endpoint `192.168.39.182` on port `80`, pointing to the Ceph object external gateway. All the other settings from the gateway section will be ignored, except for `securePort`.

Zone Settings

The zone settings allow the object store to join custom created ceph-object-zone.

- `name`: the name of the ceph-object-zone the object store will be in.

Runtime settings

MIME types

Rook provides a default `mime.types` file for each Ceph object store. This file is stored in a Kubernetes ConfigMap with the name `rook-ceph-rgw-<STORE-NAME>-mime-types`. For most users, the default file should suffice, however, the option is available to users to edit the `mime.types` file in the ConfigMap as they desire. Users may have their own special file types, and particularly security conscious users may wish to pare down the file to reduce the possibility of a file type execution attack.

Rook will not overwrite an existing `mime.types` ConfigMap so that user modifications will not be destroyed. If the object store is destroyed and recreated, the ConfigMap will also be destroyed and created anew.

Health settings

Rook-Ceph will be default monitor the state of the object store endpoints. The following CRD settings are available:

- `healthCheck`: main object store health monitoring section

Here is a complete example:

```
healthCheck:
  bucket:
    disabled: false
    interval: 60s
```

The endpoint health check procedure is the following:

1. Create an S3 user
2. Create a bucket with that user
3. PUT the file in the object store
4. GET the file from the object store
5. Verify object consistency
6. Update CR health status check

Rook-Ceph always keeps the bucket and the user for the health check, it just does a PUT and GET of an s3 object since creating a bucket is an expensive operation.

Ceph Object Multisite CRDs

The following CRDs enable Ceph object stores to isolate or replicate data via multisite. For more information on multisite, visit the ceph-object-multisite documentation.

Ceph Object Realm CRD

Rook allows creation of a realm in a ceph cluster for object stores through the custom resource definitions (CRDs). The following settings are available for Ceph object store realms.

Sample

```
apiVersion: ceph.rook.io/v1
kind: CephObjectRealm
metadata:
  name: realm-a
  namespace: rook-ceph
# This endpoint in this section needs is an endpoint from the master zone  in the master zone group of realm-a. See object-multisite.md for more details.
spec:
  pull:
    endpoint: http://10.2.105.133:80
```

Object Realm Settings

Metadata

- `name`: The name of the object realm to create
- `namespace`: The namespace of the Rook cluster where the object realm is created.

Spec

- `pull`: This optional section is for the pulling the realm for another ceph cluster.
  - `endpoint`: The endpoint in the realm from another ceph cluster you want to pull from. This endpoint must be in the master zone of the master zone group of the realm.

Ceph Object Zone Group CRD

Rook allows creation of zone groups in a ceph cluster for object stores through the custom resource definitions (CRDs). The following settings are available for Ceph

object store zone groups.

Sample

```yaml
apiVersion: ceph.rook.io/v1
kind: CephObjectZoneGroup
metadata:
  name: zonegroup-a
  namespace: rook-ceph
spec:
  realm: realm-a
```

Object Zone Group Settings

Metadata

- `name`: The name of the object zone group to create
- `namespace`: The namespace of the Rook cluster where the object zone group is created.

Spec

- `realm`: The object realm in which the zone group will be created. This matches the name of the object realm CRD.

Ceph Object Zone CRD

Rook allows creation of zones in a ceph cluster for object stores through the custom resource definitions (CRDs). The following settings are available for Ceph object store zone.

Sample

```yaml
apiVersion: ceph.rook.io/v1
kind: CephObjectZone
metadata:
  name: zone-a
  namespace: rook-ceph
spec:
  zonegroup: zonegroup-a
  metadataPool:
    failureDomain: host
    replicated:
      size: 3
  dataPool:
    failureDomain: osd
    erasureCoded:
```

```
15.       dataChunks: 2
16.       codingChunks: 1
```

Object Zone Settings

Metadata

- `name`: The name of the object zone to create
- `namespace`: The namespace of the Rook cluster where the object zone is created.

Pools

The pools allow all of the settings defined in the Pool CRD spec. For more details, see the Pool CRD settings. In the example above, there must be at least three hosts (size 3) and at least three devices (2 data + 1 coding chunks) in the cluster.

Spec

- `zonegroup`: The object zonegroup in which the zone will be created. This matches the name of the object zone group CRD.
- `metadataPool`: The settings used to create all of the object store metadata pools. Must use replication.
- `dataPool`: The settings to create the object store data pool. Can use replication or erasure coding.

Ceph Object Bucket Claim

Rook supports the creation of new buckets and access to existing buckets via two custom resources:

- an `Object Bucket Claim (OBC)` is custom resource which requests a bucket (new or existing) and is described by a Custom Resource Definition (CRD) shown below.
- an `Object Bucket (OB)` is a custom resource automatically generated when a bucket is provisioned. It is a global resource, typically not visible to non-admin users, and contains information specific to the bucket. It is described by an OB CRD, also shown below.

An OBC references a storage class which is created by an administrator. The storage class defines whether the bucket requested is a new bucket or an existing bucket. It also defines the bucket retention policy. Users request a new or existing bucket by creating an OBC which is shown below. The ceph provisioner detects the OBC and creates a new bucket or grants access to an existing bucket, depending the the storage class referenced in the OBC. It also generates a Secret which provides credentials to access the bucket, and a ConfigMap which contains the bucket's endpoint. Application pods consume the information in the Secret and ConfigMap to access the bucket. Please note that to make provisioner watch the cluster namespace only you need to set `ROOK_OBC_WATCH_OPERATOR_NAMESPACE` to `true` in the operator manifest, otherwise it watches all namespaces.

Sample

OBC Custom Resource

```
1. apiVersion: objectbucket.io/v1alpha1
2. kind: ObjectBucketClaim
3. metadata:
4.   name: ceph-bucket [1]
5.   namespace: rook-ceph [2]
6. spec:
7.   bucketName: [3]
8.   generateBucketName: photo-booth [4]
9.   storageClassName: rook-ceph-bucket [4]
10.  additionalConfig: [5]
11.    maxObjects: "1000"
12.    maxSize: "2G"
```

1. `name` of the `ObjectBucketClaim`. This name becomes the name of the Secret and ConfigMap.
2. `namespace`(optional) of the `ObjectBucketClaim`, which is also the namespace of the ConfigMap and Secret.

3. `bucketName` name of the `bucket`. **Not** recommended for new buckets since names must be unique within an entire object store.
4. `generateBucketName` value becomes the prefix for a randomly generated name, if supplied then `bucketName` must be empty. If both `bucketName` and `generateBucketName` are supplied then `BucketName` has precedence and `GenerateBucketName` is ignored. If both `bucketName` and `generateBucketName` are blank or omitted then the storage class is expected to contain the name of an *existing* bucket. It's an error if all three bucket related names are blank or omitted.
5. `storageClassName` which defines the StorageClass which contains the names of the bucket provisioner, the object-store and specifies the bucket retention policy.
6. `additionalConfig` is an optional list of key-value pairs used to define attributes specific to the bucket being provisioned by this OBC. This information is typically tuned to a particular bucket provisioner and may limit application portability. Options supported:
    - `maxObjects`: The maximum number of objects in the bucket
    - `maxSize`: The maximum size of the bucket, please note minimum recommended value is 4K.

OBC Custom Resource after Bucket Provisioning

```
1. apiVersion: objectbucket.io/v1alpha1
2. kind: ObjectBucketClaim
3. metadata:
4.   creationTimestamp: "2019-10-18T09:54:01Z"
5.   generation: 2
6.   name: ceph-bucket
7.   namespace: default [1]
8.   resourceVersion: "559491"
9. spec:
10.   ObjectBucketName: obc-default-ceph-bucket [2]
11.   additionalConfig: null
12.   bucketName: photo-booth-c1178d61-1517-431f-8408-ec4c9fa50bee [3]
13.   cannedBucketAcl: ""
14.   ssl: false
15.   storageClassName: rook-ceph-bucket [4]
16.   versioned: false
17. status:
18.   Phase: bound [5]
```

1. `namespace` where OBC got created.
2. `ObjectBucketName` generated OB name created using name space and OBC name.
3. the generated (in this case), unique `bucket name` for the new bucket.
4. name of the storage class from OBC got created.
5. phases of bucket creation:
    - *Pending*: the operator is processing the request.
    - *Bound*: the operator finished processing the request and linked the OBC and OB
    - *Released*: the OB has been deleted, leaving the OBC unclaimed but unavailable.
    - *Failed*: not currently set.

App Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
  namespace: dev-user
spec:
  containers:
  - name: mycontainer
    image: redis
    envFrom: [1]
    - configMapRef:
        name: ceph-bucket [2]
    - secretRef:
        name: ceph-bucket [3]
```

1. use `env:` if mapping of the defined key names to the env var names used by the app is needed.
2. makes available to the pod as env variables: `BUCKET_HOST`, `BUCKET_PORT`, `BUCKET_NAME`
3. makes available to the pod as env variables: `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`

StorageClass

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: rook-ceph-bucket
  labels:
    aws-s3/object [1]
provisioner: rook-ceph.ceph.rook.io/bucket [2]
parameters: [3]
  objectStoreName: my-store
  objectStoreNamespace: rook-ceph
  region: us-west-1
  bucketName: ceph-bucket [4]
reclaimPolicy: Delete [5]
```

1. `label` (optional) here associates this `StorageClass` to a specific provisioner.
2. `provisioner` responsible for handling `OBCs` referencing this `StorageClass`.
3. **all** `parameter` required.
4. `bucketName` is required for access to existing buckets but is omitted when provisioning new buckets. Unlike greenfield provisioning, the brownfield bucket name appears in the `StorageClass`, not the `OBC`.
5. rook-ceph provisioner decides how to treat the `reclaimPolicy` when an `OBC` is deleted for the bucket. See explanation as specified in Kubernetes
   - *Delete* = physically delete the bucket.
   - *Retain* = do not physically delete the bucket.

Ceph Object Store User CRD

Rook allows creation and customization of object store users through the custom resource definitions (CRDs). The following settings are available for Ceph object store users.

Sample

```yaml
apiVersion: ceph.rook.io/v1
kind: CephObjectStoreUser
metadata:
  name: my-user
  namespace: rook-ceph
spec:
  store: my-store
  displayName: my-display-name
```

Object Store User Settings

Metadata

- `name`: The name of the object store user to create, which will be reflected in the secret and other resource names.
- `namespace`: The namespace of the Rook cluster where the object store user is created.

Spec

- `store`: The object store in which the user will be created. This matches the name of the objectstore CRD.
- `displayName`: The display name which will be passed to the `radosgw-admin user create` command.

Ceph Shared Filesystem CRD

Rook allows creation and customization of shared filesystems through the custom resource definitions (CRDs). The following settings are available for Ceph filesystems.

Samples

Replicated

> **NOTE**: This sample requires *at least 1 OSD per node*, with each OSD located on *3 different nodes*.

Each OSD must be located on a different node, because both of the defined pools set the `failureDomain` to `host` and the `replicated.size` to `3`.

The `failureDomain` can also be set to another location type (e.g. `rack`), if it has been added as a `location` in the Storage Selection Settings.

```yaml
apiVersion: ceph.rook.io/v1
kind: CephFilesystem
metadata:
  name: myfs
  namespace: rook-ceph
spec:
  metadataPool:
    failureDomain: host
    replicated:
      size: 3
  dataPools:
    - failureDomain: host
      replicated:
        size: 3
  preservePoolsOnDelete: true
  metadataServer:
    activeCount: 1
    activeStandby: true
    # A key/value list of annotations
    annotations:
    #  key: value
    placement:
    #  nodeAffinity:
    #    requiredDuringSchedulingIgnoredDuringExecution:
    #      nodeSelectorTerms:
    #      - matchExpressions:
    #        - key: role
    #          operator: In
    #          values:
    #          - mds-node
```

```
    #  tolerations:
    #  - key: mds-node
    #    operator: Exists
    #  podAffinity:
    #  podAntiAffinity:
    #  topologySpreadConstraints:
    resources:
    #  limits:
    #    cpu: "500m"
    #    memory: "1024Mi"
    #  requests:
    #    cpu: "500m"
    #    memory: "1024Mi"
```

(These definitions can also be found in the `filesystem.yaml` file)

Erasure Coded

Erasure coded pools require the OSDs to use `bluestore` for the configured `storeType`. Additionally, erasure coded pools can only be used with `dataPools`. The `metadataPool` must use a replicated pool.

> **NOTE**: This sample requires *at least 3 bluestore OSDs*, with each OSD located on a *different node*.

The OSDs must be located on different nodes, because the `failureDomain` will be set to `host` by default, and the `erasureCoded` chunk settings require at least 3 different OSDs (2 `dataChunks` + 1 `codingChunks`).

```
apiVersion: ceph.rook.io/v1
kind: CephFilesystem
metadata:
  name: myfs-ec
  namespace: rook-ceph
spec:
  metadataPool:
    replicated:
      size: 3
  dataPools:
    - erasureCoded:
        dataChunks: 2
        codingChunks: 1
  metadataServer:
    activeCount: 1
    activeStandby: true
```

(These definitions can also be found in the `filesystem-ec.yaml` file)

Filesystem Settings

Metadata

- `name`: The name of the filesystem to create, which will be reflected in the pool and other resource names.
- `namespace`: The namespace of the Rook cluster where the filesystem is created.

Pools

The pools allow all of the settings defined in the Pool CRD spec. For more details, see the Pool CRD settings. In the example above, there must be at least three hosts (size 3) and at least eight devices (6 data + 2 coding chunks) in the cluster.

- `metadataPool`: The settings used to create the filesystem metadata pool. Must use replication.
- `dataPools`: The settings to create the filesystem data pools. If multiple pools are specified, Rook will add the pools to the filesystem. Assigning users or files to a pool is left as an exercise for the reader with the CephFS documentation. The data pools can use replication or erasure coding. If erasure coding pools are specified, the cluster must be running with bluestore enabled on the OSDs.
- `preservePoolsOnDelete`: If it is set to 'true' the pools used to support the filesystem will remain when the filesystem will be deleted. This is a security measure to avoid accidental loss of data. It is set to 'false' by default. If not specified is also deemed as 'false'.

Metadata Server Settings

The metadata server settings correspond to the MDS daemon settings.

- `activeCount`: The number of active MDS instances. As load increases, CephFS will automatically partition the filesystem across the MDS instances. Rook will create double the number of MDS instances as requested by the active count. The extra instances will be in standby mode for failover.
- `activeStandby`: If true, the extra MDS instances will be in active standby mode and will keep a warm cache of the filesystem metadata for faster failover. The instances will be assigned by CephFS in failover pairs. If false, the extra MDS instances will all be on passive standby mode and will not maintain a warm cache of the metadata.
- `annotations`: Key value pair list of annotations to add.
- `placement`: The mds pods can be given standard Kubernetes placement restrictions with `nodeAffinity`, `tolerations`, `podAffinity`, and `podAntiAffinity` similar to placement defined for daemons configured by the cluster CRD.
- `resources`: Set resource requests/limits for the Filesystem MDS Pod(s), see Resource Requirements/Limits.
- `priorityClassName`: Set priority class name for the Filesystem MDS Pod(s)

Ceph NFS Gateway CRD

Overview

Rook allows exporting NFS shares of the filesystem or object store through the CephNFS custom resource definition. This will spin up a cluster of NFS Ganesha servers that coordinate with one another via shared RADOS objects. The servers will be configured for NFSv4.1+ access, as serving earlier protocols can inhibit responsiveness after a server restart.

Samples

The following sample will create a two-node active-active cluster of NFS Ganesha gateways. A CephFS named `myfs` is used, and the recovery objects are stored in a RADOS pool named `myfs-data0` with a RADOS namespace of `nfs-ns`.

This example requires the filesystem to first be configured by the Filesystem.

> **NOTE**: For an RGW object store, a data pool of `my-store.rgw.buckets.data` can be used after configuring the Object Store.

```yaml
apiVersion: ceph.rook.io/v1
kind: CephNFS
metadata:
  name: my-nfs
  namespace: rook-ceph
spec:
  rados:
    # RADOS pool where NFS client recovery data is stored.
    pool: myfs-data0
    # RADOS namespace where NFS client recovery data is stored in the pool.
    namespace: nfs-ns
  # Settings for the NFS server
  server:
    # the number of active NFS servers
    active: 2
    # A key/value list of annotations
    annotations:
    #  key: value
    # where to run the NFS server
    placement:
    #  nodeAffinity:
    #    requiredDuringSchedulingIgnoredDuringExecution:
    #      nodeSelectorTerms:
    #      - matchExpressions:
    #        - key: role
    #          operator: In
```

```
    #          values:
    #          - mds-node
    #  tolerations:
    #  - key: mds-node
    #    operator: Exists
    #  podAffinity:
    #  podAntiAffinity:
    #  topologySpreadConstraints:

    # The requests and limits set here allow the ganesha pod(s) to use half of one CPU core and 1 gigabyte of memory
    resources:
    #  limits:
    #    cpu: "500m"
    #    memory: "1024Mi"
    #  requests:
    #    cpu: "500m"
    #    memory: "1024Mi"
    # the priority class to set to influence the scheduler's pod preemption
    priorityClassName:
```

NFS Settings

RADOS Settings

- `pool`: The pool where ganesha recovery backend and supplemental configuration objects will be stored
- `namespace`: The namespace in `pool` where ganesha recovery backend and supplemental configuration objects will be stored

EXPORT Block Configuration

Each daemon will have a stock configuration with no exports defined, and that includes a RADOS object via:

```
%url  rados://<pool>/<namespace>/conf-<nodeid>
```

The pool and namespace are configured via the spec’s RADOS block. The nodeid is a value automatically assigned internally by rook. Nodeids start with “a” and go through “z”, at which point they become two letters (“aa” to “az”).

When a server is started, it will create the included object if it does not already exist. It is possible to prepopulate the included objects prior to starting the server. The format for these objects is documented in the NFS Ganesha project.

Scaling the active server count

It is possible to scale the size of the cluster up or down by modifying the `spec.server.active` field. Scaling the cluster size up can be done at will. Once the new server comes up, clients can be assigned to it immediately.

The CRD always eliminates the highest index servers first, in reverse order from how they were started. Scaling down the cluster requires that clients be migrated from servers that will be eliminated to others. That process is currently a manual one and should be performed before reducing the size of the cluster.

Ceph CSI Drivers

There are two CSI drivers integrated with Rook that will enable different scenarios:

- RBD: This driver is optimized for RWO pod access where only one pod may access the storage
- CephFS: This driver allows for RWX with one or more pods accessing the same storage

The drivers are enabled automatically with the Rook operator. They will be started in the same namespace as the operator when the first CephCluster CR is created.

For documentation on consuming the storage:

- RBD: See the Block Storage topic
- CephFS: See the Shared Filesystem topic

Configure CSI Drivers in non-default namespace

If you've deployed the Rook operator in a namespace other than "rook-ceph", change the prefix in the provisioner to match the namespace you used. For example, if the Rook operator is running in the namespace "my-namespace" the provisioner value should be "my-namespace.rbd.csi.ceph.com". The same provisioner name needs to be set in both the storageclass and snapshotclass.

Liveness Sidecar

All CSI pods are deployed with a sidecar container that provides a prometheus metric for tracking if the CSI plugin is alive and runnning. These metrics are meant to be collected by prometheus but can be acceses through a GET request to a specific node ip. for example `curl -X get http://[pod ip]:[liveness-port][liveness-path] 2>/dev/null | grep csi` the expected output should be

```
$ curl -X GET http://10.109.65.142:9080/metrics 2>/dev/null | grep csi
# HELP csi_liveness Liveness Probe
# TYPE csi_liveness gauge
csi_liveness 1
```

Check the monitoring doc to see how to integrate CSI liveness and grpc metrics into ceph monitoring.

Dynamically Expand Volume

Prerequisite

- For filesystem resize to be supported for your Kubernetes cluster, the kubernetes version running in your cluster should be >= v1.15 and for block volume resize support the Kubernetes version should be >= v1.16. Also, `ExpandCSIVolumes` feature gate has to be enabled for the volume resize functionality to work.

To expand the PVC the controlling StorageClass must have `allowVolumeExpansion` set to `true` . `csi.storage.k8s.io/controller-expand-secret-name` and `csi.storage.k8s.io/controller-expand-secret-namespace` values set in storageclass. Now expand the PVC by editing the PVC `pvc.spec.resource.requests.storage` to a higher values than the current size. Once PVC is expanded on backend and same is reflected size is reflected on application mountpoint, the status capacity `pvc.status.capacity.storage` of PVC will be updated to new size.

Prerequisites

- Requires Kubernetes v1.17+ which supports snapshot beta.
- Install the new snapshot controller and snapshot beta CRD. More info can be found here

Note: If the Kubernetes distributor you are using does not supports the snapshot beta, still you can use the Alpha snapshots. refer to snapshot on how to use snapshots.

- We also need a `SnapshotClass` for volume snapshot to work. The purpose of a `SnapshotClass` is defined in the kubernetes documentation. In short, as the documentation describes it:

> Just like StorageClass provides a way for administrators to describe the “classes” of storage they offer when provisioning a volume, VolumeSnapshotClass provides a way to describe the “classes” of storage when provisioning a volume snapshot.

RBD Snapshots

SnapshotClass

In snapshotClass, the `csi.storage.k8s.io/snapshotter-secret-name` parameter should reference the name of the secret created for the rbdplugin and `pool` to reflect the Ceph pool name.

Update the value of the `clusterID` field to match the namespace that Rook is running in. When Ceph CSI is deployed by Rook, the operator will automatically maintain a configmap whose contents will match this key. By default this is “rook-ceph”.

```
1. kubectl create -f cluster/examples/kubernetes/ceph/csi/rbd/snapshotclass.yaml
```

Volumesnapshot

In snapshot, `volumeSnapshotClassName` should be the name of the `VolumeSnapshotClass` previously created. The `persistentVolumeClaimName` should be the name of the PVC which is already created by the RBD CSI driver.

```
1. kubectl create -f cluster/examples/kubernetes/ceph/csi/rbd/snapshot.yaml
```

Verify RBD Snapshot Creation

```
1. kubectl get volumesnapshotclass
2. NAME                      DRIVER                       DELETIONPOLICY   AGE
3. csi-rbdplugin-snapclass   rook-ceph.rbd.csi.ceph.com   Delete           3h55m
```

```
kubectl get volumesnapshot
NAME                READYTOUSE   SOURCEPVC   SOURCESNAPSHOTCONTENT   RESTORESIZE   SNAPSHOTCLASS
SNAPSHOTCONTENT                                    CREATIONTIME   AGE
rbd-pvc-snapshot    true         rbd-pvc                             1Gi           csi-rbdplugin-snapclass
snapcontent-79090db0-7c66-4b18-bf4a-634772c7cac7   3h50m          3h51m
```

The snapshot will be ready to restore to a new PVC when the `READYTOUSE` field of the `volumesnapshot` is set to true.

Restore the snapshot to a new PVC

In pvc-restore, `dataSource` should be the name of the `VolumeSnapshot` previously created. The `dataSource` kind should be the `VolumeSnapshot`.

Create a new PVC from the snapshot

```
kubectl create -f cluster/examples/kubernetes/ceph/csi/rbd/pvc-restore.yaml
```

Verify RBD Clone PVC Creation

```
kubectl get pvc
NAME              STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS
AGE
rbd-pvc           Bound    pvc-84294e34-577a-11e9-b34f-525400581048   1Gi        RWO            rook-ceph-
block       34m
rbd-pvc-restore   Bound    pvc-575537bf-577f-11e9-b34f-525400581048   1Gi        RWO            rook-ceph-
block       8s
```

RBD snapshot resource Cleanup

To clean your cluster of the resources created by this example, run the following:

```
kubectl delete -f cluster/examples/kubernetes/ceph/csi/rbd/pvc-restore.yaml
kubectl delete -f cluster/examples/kubernetes/ceph/csi/rbd/snapshot.yaml
kubectl delete -f cluster/examples/kubernetes/ceph/csi/rbd/snapshotclass.yaml
```

CephFS Snapshots

SnapshotClass

In snapshotClass, the `csi.storage.k8s.io/snapshotter-secret-name` parameter should reference the name of the secret created for the cephfsplugin.

In the snapshotclass, update the value of the `clusterID` field to match the namespace that Rook is running in. When Ceph CSI is deployed by Rook, the operator will

automatically maintain a configmap whose contents will match this key. By default this is “rook-ceph”.

```
kubectl create -f cluster/examples/kubernetes/ceph/csi/cephfs/snapshotclass.yaml
```

Volumesnapshot

In snapshot, `volumeSnapshotClassName` should be the name of the `VolumeSnapshotClass` previously created. The `persistentVolumeClaimName` should be the name of the PVC which is already created by the CephFS CSI driver.

```
kubectl create -f cluster/examples/kubernetes/ceph/csi/cephfs/snapshot.yaml
```

Verify CephFS Snapshot Creation

```
kubectl get volumesnapshotclass
NAME                        DRIVER                          DELETIONPOLICY   AGE
csi-cephfslugin-snapclass   rook-ceph.cephfs.csi.ceph.com   Delete           3h55m

kubectl get volumesnapshot
NAME                  READYTOUSE   SOURCEPVC   SOURCESNAPSHOTCONTENT   RESTORESIZE   SNAPSHOTCLASS
SNAPSHOTCONTENT                                    CREATIONTIME   AGE
cephfs-pvc-snapshot   true         cephfs-pvc                          1Gi           csi-cephfsplugin-snapclass
snapcontent-34476204-a14a-4d59-bfbc-2bbba695652c   3h50m          3h51m
```

The snapshot will be ready to restore to a new PVC when `READYTOUSE` field of the `volumesnapshot` is set to true.

Restore the snapshot to a new PVC

In pvc-restore, `dataSource` should be the name of the `VolumeSnapshot` previously created. The `dataSource` kind should be the `VolumeSnapshot` .

Create a new PVC from the snapshot

```
kubectl create -f cluster/examples/kubernetes/ceph/csi/cephfs/pvc-restore.yaml
```

Verify CephFS Restore PVC Creation

```
kubectl get pvc
NAME                STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS      AGE
cephfs-pvc          Bound    pvc-74734901-577a-11e9-b34f-525400581048   1Gi        RWX            rook-cephfs
55m
cephfs-pvc-restore   Bound    pvc-95308c75-6c93-4928-a551-6b5137192209   1Gi        RWX            rook-cephfs
34s
```

CephFS snapshot resource Cleanup

To clean your cluster of the resources created by this example, run the following:

```
kubectl delete -f cluster/examples/kubernetes/ceph/csi/cephfs/pvc-restore.yaml
kubectl delete -f cluster/examples/kubernetes/ceph/csi/cephfs/snapshot.yaml
kubectl delete -f cluster/examples/kubernetes/ceph/csi/cephfs/snapshotclass.yaml
```

Limitations

- There is a limit of 400 snapshots per cephFS filesystem.
- The PVC cannot be deleted if it has snapshots. make sure all the snapshots on the PVC are deleted before you delete the PVC.

The CSI Volume Cloning feature adds support for specifying existing PVCs in the `dataSource` field to indicate a user would like to clone a Volume.

A Clone is defined as a duplicate of an existing Kubernetes Volume that can be consumed as any standard Volume would be. The only difference is that upon provisioning, rather than creating a "new" empty Volume, the back end device creates an exact duplicate of the specified Volume.

Refer to clone-doc for more info.

RBD Volume Cloning

Volume Clone Prerequisites

1. Requires Kubernetes v1.16+ which supports volume clone.
2. Ceph-csi diver v3.0.0+ which supports volume clone.

Volume Cloning

In pvc-clone, `dataSource` should be the name of the `PVC` which is already created by RBD CSI driver. The `dataSource` kind should be the `PersistentVolumeClaim` and also storageclass should be same as the source `PVC` .

Create a new PVC Clone from the PVC

```
kubectl create -f cluster/examples/kubernetes/ceph/csi/rbd/pvc-clone.yaml
```

Verify RBD volume Clone PVC Creation

```
kubectl get pvc
NAME              STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS
AGE
rbd-pvc           Bound    pvc-74734901-577a-11e9-b34f-525400581048   1Gi        RWO            rook-ceph-
block       34m
rbd-pvc-clone     Bound    pvc-70473135-577f-11e9-b34f-525400581048   1Gi        RWO            rook-ceph-
block       8s
```

RBD clone resource Cleanup

To clean your cluster of the resources created by this example, run the following:

```
kubectl delete -f cluster/examples/kubernetes/ceph/csi/rbd/pvc-clone.yaml
```

CephFS Volume Cloning

Volume Clone Prerequisites

1. Requires Kubernetes v1.16+ which supports volume clone.
2. Ceph-csi diver v3.1.0+ which supports volume clone.

Volume Cloning

In pvc-clone, `dataSource` should be the name of the `PVC` which is already created by CephFS CSI driver. The `dataSource` kind should be the `PersistentVolumeClaim` and also storageclass should be same as the source `PVC` .

Create a new PVC Clone from the PVC

```
kubectl create -f cluster/examples/kubernetes/ceph/csi/cephfs/pvc-clone.yaml
```

Verify CephFS volume Clone PVC Creation

```
kubectl get pvc
NAME              STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS
AGE
cephfs-pvc        Bound    pvc-1ea51547-a88b-4ab0-8b4a-812caeaf025d   1Gi        RWX            rook-cephfs
39m
cephfs-pvc-clone  Bound    pvc-b575bc35-d521-4c41-b4f9-1d733cd28fdf   1Gi        RWX            rook-cephfs
8s
```

CephFS clone resource Cleanup

To clean your cluster of the resources created by this example, run the following:

```
kubectl delete -f cluster/examples/kubernetes/ceph/csi/cephfs/pvc-clone.yaml
```

Ceph Client CRD

Rook allows creation and updating clients through the custom resource definitions (CRDs). For more information about user management and capabilities see the Ceph docs.

Use Case

Use Client CRD in case you want to integrate Rook with with applications that are using LibRBD directly. For example for OpenStack deployment with Ceph backend use Client CRD to create OpenStack services users.

The Client CRD is not needed for Flex or CSI driver users. The drivers create the needed users automatically.

Creating Ceph User

To get you started, here is a simple example of a CRD to configure a Ceph client with capabilities.

```
apiVersion: ceph.rook.io/v1
kind: CephClient
metadata:
  name: glance
  namespace: rook-ceph
spec:
  caps:
    mon: 'profile rbd'
    osd: 'profile rbd pool=images'
---
apiVersion: ceph.rook.io/v1
kind: CephClient
metadata:
  name: cinder
  namespace: rook-ceph
spec:
  caps:
    mon: 'profile rbd'
    osd: 'profile rbd pool=volumes, profile rbd pool=vms, profile rbd-read-only pool=images'
```

Prerequisites

This guide assumes you have created a Rook cluster as explained in the main Quickstart guide

Ceph RBDMirror CRD

Rook allows creation and updating rbd-mirror daemon(s) through the custom resource definitions (CRDs). RBD images can be asynchronously mirrored between two Ceph clusters. For more information about user management and capabilities see the Ceph docs.

Creating daemons

To get you started, here is a simple example of a CRD to deploy an rbd-mirror daemon.

```
1. apiVersion: ceph.rook.io/v1
2. kind: CephRBDMirror
3. metadata:
4.   name: my-rbd-mirror
5.   namespace: rook-ceph
6. spec:
7.   count: 1
```

Prerequisites

This guide assumes you have created a Rook cluster as explained in the main Quickstart guide

Configuration

For most any Ceph cluster, the user will want to–and may need to–change some Ceph configurations. These changes often may be warranted in order to alter performance to meet SLAs or to update default data resiliency settings.

> **WARNING**: Modify Ceph settings carefully, and review the Ceph configuration documentation before making any changes. Changing the settings could result in unhealthy daemons or even data loss if used incorrectly.

Required configurations

Rook and Ceph both strive to make configuration as easy as possible, but there are some configuration options which users are well advised to consider for any production cluster.

Default PG and PGP counts

The number of PGs and PGPs can be configured on a per-pool basis, but it is highly advised to set default values that are appropriate for your Ceph cluster. Appropriate values depend on the number of OSDs the user expects to have backing each pool. The Ceph OSD and Pool config docs provide detailed information about how to tune these parameters: `osd_pool_default_pg_num` and `osd_pool_default_pgp_num`.

Pools created prior to v1.1 will have a default PG count of 100. Pools created after v1.1 will have Ceph's default PG count.

An easier option exists for Rook-Ceph clusters running Ceph Nautilus (v14.2.x) or newer. Nautilus introduced the PG auto-scaler mgr module capable of automatically managing PG and PGP values for pools. Please see Ceph New in Nautilus: PG merging and autotuning for more information about this module.

In Nautilus, This module is not enabled by default, but can be enabled by the following setting in the CephCluster CR:

```
spec:
  mgr:
    modules:
    - name: pg_autoscaler
      enabled: true
```

In Octopus (v15.2.x), this module is enabled by default without the above-mentioned setting.

With that setting, the autoscaler will be enabled for all new pools. If you do not desire to have the autoscaler enabled for all new pools, you will need to use the Rook

toolbox to enable the module and enable the autoscaling on individual pools.

The autoscaler is not enabled for the existing pools after enabling the module. So if you want to enable the autoscaling for these existing pools, they must be configured from the toolbox.

Specifying configuration options

Toolbox + Ceph CLI

The most recommended way of configuring Ceph is to set Ceph's configuration directly. The first method for doing so is to use Ceph's CLI from the Rook-Ceph toolbox pod. Using the toolbox pod is detailed here. From the toolbox, the user can change Ceph configurations, enable manager modules, create users and pools, and much more.

Ceph Dashboard

The Ceph Dashboard, examined in more detail here, is another way of setting some of Ceph's configuration directly. Configuration by the Ceph dashboard is recommended with the same priority as configuration via the Ceph CLI (above).

Advanced configuration via ceph.conf override ConfigMap

Setting configs via Ceph's CLI requires that at least one mon be available for the configs to be set, and setting configs via dashboard requires at least one mgr to be available. Ceph may also have a small number of very advanced settings that aren't able to be modified easily via CLI or dashboard. The **least** recommended method for configuring Ceph is intended as a last-resort fallback in situations like these. This is covered in detail here.

Rook-Ceph Upgrades

This guide will walk you through the steps to upgrade the software in a Rook-Ceph cluster from one version to the next. This includes both the Rook-Ceph operator software itself as well as the Ceph cluster software.

Upgrades for both the operator and for Ceph are nearly entirely automated save for where Rook's permissions need to be explicitly updated by an admin or when incompatibilities need to be addressed manually due to customizations.

We welcome feedback and opening issues!

Supported Versions

This guide is for upgrading from **Rook v1.3.x to Rook v1.4.x**.

Please refer to the upgrade guides from previous releases for supported upgrade paths. Rook upgrades are only supported between official releases. Upgrades to and from `master` are not supported.

For a guide to upgrade previous versions of Rook, please refer to the version of documentation for those releases.

- Upgrade 1.2 to 1.3
- Upgrade 1.1 to 1.2
- Upgrade 1.0 to 1.1
- Upgrade 0.9 to 1.0
- Upgrade 0.8 to 0.9
- Upgrade 0.7 to 0.8
- Upgrade 0.6 to 0.7
- Upgrade 0.5 to 0.6

Considerations

With this upgrade guide, there are a few notes to consider:

- **WARNING**: Upgrading a Rook cluster is not without risk. There may be unexpected issues or obstacles that damage the integrity and health of your storage cluster, including data loss.
- The Rook cluster's storage may be unavailable for short periods during the upgrade process for both Rook operator updates and for Ceph version updates.
- We recommend that you read this document in full before you undertake a Rook cluster upgrade.

Before you Upgrade

Rook v1.4 has a breaking change that should be considered before upgrading.

1. Make sure the `lvm2` package is installed on the host where OSDs are running. If not, the prepare job will fail and the upgrade of OSDs will be blocked.
    - This only applies to non-PVC OSDs. Beginning in v1.3, OSDs created on PVCs no longer rely on LVM.
2. CSI Snapshots: See the next section if you are currently using Ceph-CSI snapshots.

CSI Snapshots

CSI snapshots have moved from Alpha to Beta and are **not** backward compatible. The snapshots created with the Alpha version must be deleted before the upgrade.

If you desire to continue using Ceph-CSI 2.x with Alpha snapshots, then:

- Skip this section
- Set the CSI version to 2.x as described below in the CSI Version section.

To continue with the Ceph-CSI v3.0 driver that is recommended with v1.4:

1. List all the volumesnapshots created

```
kubectl get volumesnapshot
NAME               AGE
rbd-pvc-snapshot   22s
```

1. Delete all volumesnapshots

```
kubectl delete volumesnapshot rbd-pvc-snapshot
volumesnapshot.snapshot.storage.k8s.io "rbd-pvc-snapshot" deleted
```

1. List all volumesnapshotclasses created

```
kubectl get volumesnapshotclass
NAME                      AGE
csi-rbdplugin-snapclass   86s
```

1. Delete all volumesnapshotclasses

```
kubectl delete volumesnapshotclass csi-rbdplugin-snapclass
volumesnapshotclass.snapshot.storage.k8s.io "csi-rbdplugin-snapclass" deleted
```

Note: The underlying snapshots on the storage system will be deleted by ceph-csi

Delete the Alpha CRDs

As we are updating the snapshot resources from `Alpha` to `Beta` we need to delete the old `alphav1` snapshot CRD created by external-snapshotter sidecar container

Check if we have any `v1alpha1` CRD created in our kubernetes cluster

```
kubectl get crd volumesnapshotclasses.snapshot.storage.k8s.io -o yaml |grep v1alpha1
  - name: v1alpha1
  - v1alpha1
kubectl get crd volumesnapshotcontents.snapshot.storage.k8s.io -o yaml |grep v1alpha1
  - name: v1alpha1
  - v1alpha1
kubectl get crd volumesnapshots.snapshot.storage.k8s.io -o yaml |grep v1alpha1
  - name: v1alpha1
  - v1alpha1
```

As we have `v1alpha1` CRD created in our kubernetes cluster, we need to delete the Alpha CRD

```
   kubectl delete crd volumesnapshotclasses.snapshot.storage.k8s.io
volumesnapshotcontents.snapshot.storage.k8s.io volumesnapshots.snapshot.storage.k8s.io

customresourcedefinition.apiextensions.k8s.io "volumesnapshotclasses.snapshot.storage.k8s.io" deleted
customresourcedefinition.apiextensions.k8s.io "volumesnapshotcontents.snapshot.storage.k8s.io" deleted
customresourcedefinition.apiextensions.k8s.io "volumesnapshots.snapshot.storage.k8s.io" deleted
```

Finally, if you desire to use the beta snapshots, check that the prerequisites are met.

Upgrading the Rook-Ceph Operator

Patch Release Upgrades

Unless otherwise noted due to extenuating requirements, upgrades from one patch release of Rook to another are as simple as updating the image of the Rook operator. For example, when Rook v1.4.1 is released, the process of updating from v1.4.1 is as simple as running the following:

```
kubectl -n rook-ceph set image deploy/rook-ceph-operator rook-ceph-operator=rook/ceph:v1.4.1
```

Helm Upgrades

If you have installed Rook via the Helm chart, Helm will handle some details of the upgrade for you. In particular, Helm will handle updating the RBAC and trigger the operator update and restart.

Upgrading from v1.3 to v1.4

Rook releases from master are expressly unsupported. It is strongly recommended that you use official releases of Rook. Unreleased versions from the master branch are subject to changes and incompatibilities that will not be supported in the official releases. Builds from the master branch can have functionality changed and even removed at any time without compatibility support and without prior notice.

Prerequisites

We will do all our work in the Ceph example manifests directory.

```
cd $YOUR_ROOK_REPO/cluster/examples/kubernetes/ceph/
```

Unless your Rook cluster was created with customized namespaces, namespaces for Rook clusters created before v0.8 are likely to be:

- Clusters created by v0.7 or earlier: `rook-system` and `rook`
- Clusters created in v0.8 or v0.9: `rook-ceph-system` and `rook-ceph`
- Clusters created in v1.0 or newer: only `rook-ceph`

With this guide, we do our best not to assume the namespaces in your cluster. To make things as easy as possible, modify and use the below snippet to configure your environment. We will use these environment variables throughout this document.

```
# Parameterize the environment
export ROOK_SYSTEM_NAMESPACE="rook-ceph"
export ROOK_NAMESPACE="rook-ceph"
```

In order to successfully upgrade a Rook cluster, the following prerequisites must be met:

- The cluster should be in a healthy state with full functionality. Review the health verification section in order to verify your cluster is in a good starting state.
- All pods consuming Rook storage should be created, running, and in a steady state. No Rook persistent volumes should be in the act of being created or deleted.

Helm

- Your Helm version should be newer than v3.2.0 to avoid this issue.
- For a Rook cluster already deployed with Helm older than v3.2.0, also execute the following commands.

```
KIND=ClusterRole
```

```
NAME=psp:rook
RELEASE=your-apps-release-name
NAMESPACE=your-apps-namespace
kubectl annotate $KIND $NAME meta.helm.sh/release-name=$RELEASE
kubectl annotate $KIND $NAME meta.helm.sh/release-namespace=$NAMESPACE
kubectl label $KIND $NAME app.kubernetes.io/managed-by=Helm
```

Health Verification

Before we begin the upgrade process, let's first review some ways that you can verify the health of your cluster, ensuring that the upgrade is going smoothly after each step. Most of the health verification checks for your cluster during the upgrade process can be performed with the Rook toolbox. For more information about how to run the toolbox, please visit the Rook toolbox readme.

See the common issues pages for troubleshooting and correcting health issues:

- General troubleshooting
- Ceph troubleshooting

Pods all Running

In a healthy Rook cluster, the operator, the agents and all Rook namespace pods should be in the `Running` state and have few, if any, pod restarts. To verify this, run the following commands:

```
kubectl -n $ROOK_NAMESPACE get pods
```

Status Output

The Rook toolbox contains the Ceph tools that can give you status details of the cluster with the `ceph status` command. Let's look at an output sample and review some of the details:

```
TOOLS_POD=$(kubectl -n $ROOK_NAMESPACE get pod -l "app=rook-ceph-tools" -o
jsonpath='{.items[0].metadata.name}')
kubectl -n $ROOK_NAMESPACE exec -it $TOOLS_POD -- ceph status
```

```
  cluster:
    id:     a3f4d647-9538-4aff-9fd1-b845873c3fe9
    health: HEALTH_OK

  services:
    mon: 3 daemons, quorum b,c,a
    mgr: a(active)
    mds: myfs-1/1/1 up  {0=myfs-a=up:active}, 1 up:standby-replay
    osd: 6 osds: 6 up, 6 in
```

```
    rgw: 1 daemon active

  data:
    pools:   9 pools, 900 pgs
    objects: 67  objects, 11 KiB
    usage:   6.1 GiB used, 54 GiB / 60 GiB avail
    pgs:     900 active+clean

  io:
    client:   7.4 KiB/s rd, 681 B/s wr, 11 op/s rd, 4 op/s wr
    recovery: 164 B/s, 1 objects/s
```

In the output above, note the following indications that the cluster is in a healthy state:

- Cluster health: The overall cluster status is `HEALTH_OK` and there are no warning or error status messages displayed.
- Monitors (mon): All of the monitors are included in the `quorum` list.
- Manager (mgr): The Ceph manager is in the `active` state.
- OSDs (osd): All OSDs are `up` and `in`.
- Placement groups (pgs): All PGs are in the `active+clean` state.
- (If applicable) Ceph filesystem metadata server (mds): all MDSes are `active` for all filesystems
- (If applicable) Ceph object store RADOS gateways (rgw): all daemons are `active`

If your `ceph status` output has deviations from the general good health described above, there may be an issue that needs to be investigated further. There are other commands you may run for more details on the health of the system, such as `ceph osd status`. See the Ceph troubleshooting docs for help.

Rook will prevent the upgrade of the Ceph daemons if the health is in a `HEALTH_ERR` state. If you desired to proceed with the upgrade anyway, you will need to set either `skipUpgradeChecks: true` or `continueUpgradeAfterChecksEvenIfNotHealthy: true` as described in the cluster CR settings.

Container Versions

The container version running in a specific pod in the Rook cluster can be verified in its pod spec output. For example for the monitor pod `mon-b`, we can verify the container version it is running with the below commands:

```
POD_NAME=$(kubectl -n $ROOK_NAMESPACE get pod -o custom-columns=name:.metadata.name --no-headers | grep rook-ceph-mon-a)
kubectl -n $ROOK_NAMESPACE get pod ${POD_NAME} -o jsonpath='{.spec.containers[0].image}'
```

The status and container versions for all Rook pods can be collected all at once with the following commands:

```
kubectl -n $ROOK_SYSTEM_NAMESPACE get pod -o jsonpath='{range .items[*]}{.metadata.name}{"\n\t"}
{.status.phase}{"\t\t"}{.spec.containers[0].image}{"\t"}{.spec.initContainers[0]}{"\n"}{end}' && \
kubectl -n $ROOK_NAMESPACE get pod -o jsonpath='{range .items[*]}{.metadata.name}{"\n\t"}{.status.phase}
{"\t\t"}{.spec.containers[0].image}{"\t"}{.spec.initContainers[0].image}{"\n"}{end}'
```

The `rook-version` label exists on Ceph controller resources. For various resource controllers, a summary of the resource controllers can be gained with the commands below. These will report the requested, updated, and currently available replicas for various Rook-Ceph resources in addition to the version of Rook for resources managed by the updated Rook-Ceph operator. Note that the operator and toolbox deployments do not have a `rook-version` label set.

```
kubectl -n $ROOK_NAMESPACE get deployments -o jsonpath='{range .items[*]}{.metadata.name}{"  \treq/upd/avl: "}
{.spec.replicas}{"/"}{.status.updatedReplicas}{"/"}{.status.readyReplicas}{"  \trook-version="}
{.metadata.labels.rook-version}{"\n"}{end}'

kubectl -n $ROOK_NAMESPACE get jobs -o jsonpath='{range .items[*]}{.metadata.name}{"  \tsucceeded: "}
{.status.succeeded}{"      \trook-version="}{.metadata.labels.rook-version}{"\n"}{end}'
```

Rook Volume Health

Any pod that is using a Rook volume should also remain healthy:

- The pod should be in the `Running` state with few, if any, restarts
- There should be no errors in its logs
- The pod should still be able to read and write to the attached Rook volume.

Rook Operator Upgrade Process

In the examples given in this guide, we will be upgrading a live Rook cluster running `v1.3.9` to the version `v1.4.1`. This upgrade should work from any official patch release of Rook v1.3 to any official patch release of v1.4.

Rook release from `master` are expressly unsupported. It is strongly recommended that you use official releases of Rook. Unreleased versions from the master branch are subject to changes and incompatibilities that will not be supported in the official releases. Builds from the master branch can have functionality changed or removed at any time without compatibility support and without prior notice.

Let’s get started!

1. Remove Alpha Snapshots

As described above in the CSI Snapshots section, if you are using alpha snapshots it is required to remove them before the upgrade since they are not compatible with beta snaphots available with Ceph-CSI v3.0.

2. Update the RBAC and CRDs

> Automatically updated if you are upgrading via the helm chart

First apply new resources. This includes modified privileges (RBAC) needed by the Operator and updates to the Custom Resource Definitions (CRDs).

If you are not using the default `rook-ceph` namespace, replace the namespace in the following manifest:

```
1. sed -i "s/namespace: rook-ceph/namespace: $ROOK_SYSTEM_NAMESPACE/g" upgrade-from-v1.3-apply.yaml
```

Now apply the updated privileges:

```
1. kubectl delete -f upgrade-from-v1.3-delete.yaml
2. kubectl apply -f upgrade-from-v1.3-apply.yaml -f upgrade-from-v1.3-crds.yaml
```

3. Update Ceph CSI version to v3.0

> Rook v1.4 will install Ceph-CSI v3.0 use the latest drivers by default. If you have not specified custom CSI images in the Operator deployment this step is unnecessary.

If you have specified custom CSI images in the Rook-Ceph Operator deployment, it is recommended to update to use the latest Ceph-CSI v3.0 driver. See the section CSI Version for more details.

4. Update the Rook Operator

> Automatically updated if you are upgrading via the helm chart

The largest portion of the upgrade is triggered when the operator's image is updated to `v1.4.x`. When the operator is updated, it will proceed to update all of the Ceph daemons.

```
1. kubectl -n $ROOK_SYSTEM_NAMESPACE set image deploy/rook-ceph-operator rook-ceph-operator=rook/ceph:v1.4.1
```

5. Wait for the upgrade to complete

Watch now in amazement as the Ceph mons, mgrs, OSDs, rbd-mirrors, MDSes and RGWs are terminated and replaced with updated versions in sequence. The cluster may be offline very briefly as mons update, and the Ceph Filesystem may fall offline a few times while the MDSes are upgrading. This is normal.

The versions of the components can be viewed as they are updated:

```
    watch --exec kubectl -n $ROOK_NAMESPACE get deployments -l rook_cluster=$ROOK_NAMESPACE -o jsonpath='{range
    .items[*]}{.metadata.name}{"  \treq/upd/avl: "}{.spec.replicas}{"/"}{.status.updatedReplicas}{"/"}
1.  {.status.readyReplicas}{"  \trook-version="}{.metadata.labels.rook-version}{"\n"}{end}'
```

As an example, this cluster is midway through updating the OSDs from v1.3 to v1.4. When all deployments report `1/1/1` availability and `rook-version=v1.4.1`, the Ceph cluster's core components are fully updated.

```
1.  Every 2.0s: kubectl -n rook-ceph get deployment -o j...
2.
3.  rook-ceph-mgr-a         req/upd/avl: 1/1/1      rook-version=v1.4.1
4.  rook-ceph-mon-a         req/upd/avl: 1/1/1      rook-version=v1.4.1
5.  rook-ceph-mon-b         req/upd/avl: 1/1/1      rook-version=v1.4.1
6.  rook-ceph-mon-c         req/upd/avl: 1/1/1      rook-version=v1.4.1
7.  rook-ceph-osd-0         req/upd/avl: 1//        rook-version=v1.4.1
8.  rook-ceph-osd-1         req/upd/avl: 1/1/1      rook-version=v1.3.9
9.  rook-ceph-osd-2         req/upd/avl: 1/1/1      rook-version=v1.3.9
```

An easy check to see if the upgrade is totally finished is to check that there is only one `rook-version` reported across the cluster.

```
    # kubectl -n $ROOK_NAMESPACE get deployment -l rook_cluster=$ROOK_NAMESPACE -o jsonpath='{range .items[*]}
1.  {"rook-version="}{.metadata.labels.rook-version}{"\n"}{end}' | sort | uniq
2.  This cluster is not yet finished:
3.    rook-version=v1.3.9
4.    rook-version=v1.4.1
5.  This cluster is finished:
6.    rook-version=v1.4.1
```

6. Verify the updated cluster

At this point, your Rook operator should be running version `rook/ceph:v1.4.1`.

Verify the Ceph cluster's health using the health verification section.

Ceph Version Upgrades

Rook v1.4 supports Ceph Nautilus 14.2.5 or newer and Ceph Octopus v15.2.0 or newer. These are the only supported major versions of Ceph.

> IMPORTANT: When an update is requested, the operator will check Ceph's status, if it is in `HEALTH_ERR` it will refuse to do the upgrade.

Rook is cautious when performing upgrades. When an upgrade is requested (the Ceph image has been updated in the CR), Rook will go through all the daemons one by one and will individually perform checks on them. It will make sure a particular daemon can be stopped before performing the upgrade. Once the deployment has been updated, it checks if this is ok to continue. After each daemon is updated we wait for things to settle

(monitors to be in a quorum, PGs to be clean for OSDs, up for MDSs, etc.), then only when the condition is met we move to the next daemon. We repeat this process until all the daemons have been updated.

Ceph images

Official Ceph container images can be found on Docker Hub. These images are tagged in a few ways:

- The most explicit form of tags are full-ceph-version-and-build tags (e.g., `v15.2.4-20200630`). These tags are recommended for production clusters, as there is no possibility for the cluster to be heterogeneous with respect to the version of Ceph running in containers.
- Ceph major version tags (e.g., `v15`) are useful for development and test clusters so that the latest version of Ceph is always available.

Ceph containers other than the official images from the registry above will not be supported.

Example upgrade to Ceph Octopus

1. Update the main Ceph daemons

The majority of the upgrade will be handled by the Rook operator. Begin the upgrade by changing the Ceph image field in the cluster CRD (`spec.cephVersion.image`).

```
NEW_CEPH_IMAGE='ceph/ceph:v15.2.4-20200630'
CLUSTER_NAME="$ROOK_NAMESPACE"  # change if your cluster name is not the Rook namespace
kubectl -n $ROOK_NAMESPACE patch CephCluster $CLUSTER_NAME --type=merge -p "{\"spec\": {\"cephVersion\":
{\"image\": \"$NEW_CEPH_IMAGE\"}}}"
```

2. Wait for the daemon pod updates to complete

As with upgrading Rook, you must now wait for the upgrade to complete. Status can be determined in a similar way to the Rook upgrade as well.

```
watch --exec kubectl -n $ROOK_NAMESPACE get deployments -l rook_cluster=$ROOK_NAMESPACE -o jsonpath='{range
.items[*]}{.metadata.name}{"  \treq/upd/avl: "}{.spec.replicas}{"/"}{.status.updatedReplicas}{"/"}
{.status.readyReplicas}{"  \tceph-version="}{.metadata.labels.ceph-version}{"\n"}{end}'
```

Determining when the Ceph has fully updated is rather simple.

```
# kubectl -n $ROOK_NAMESPACE get deployment -l rook_cluster=$ROOK_NAMESPACE -o jsonpath='{range .items[*]}
{"ceph-version="}{.metadata.labels.ceph-version}{"\n"}{end}' | sort | uniq
This cluster is not yet finished:
    ceph-version=14.2.7-0
    ceph-version=15.2.4-0
This cluster is finished:
```

```
    ceph-version=15.2.4-0
```

3. Verify the updated cluster

Verify the Ceph cluster's health using the health verification section.

CSI Version

If you have a cluster running with CSI drivers enabled and you want to configure Rook to use non-default CSI images, the following settings will need to be applied for the desired version of CSI.

The operator configuration variables have recently moved from the operator deployment to the `rook-ceph-operator-config` ConfigMap. The values in the operator deployment can still be set, but if the ConfigMap settings are applied, they will override the operator deployment settings.

> If the cluster was originally installed prior to v1.3, the configmap may not exist. See the latest operator.yaml for example configmap settings.

```
kubectl -n $ROOK_NAMESPACE edit configmap rook-ceph-operator-config
```

The default upstream images are included below, which you can change to your desired images.

```
ROOK_CSI_CEPH_IMAGE: "quay.io/cephcsi/cephcsi:v3.1.0"
ROOK_CSI_REGISTRAR_IMAGE: "quay.io/k8scsi/csi-node-driver-registrar:v1.2.0"
ROOK_CSI_PROVISIONER_IMAGE: "quay.io/k8scsi/csi-provisioner:v1.6.0"
ROOK_CSI_SNAPSHOTTER_IMAGE: "quay.io/k8scsi/csi-snapshotter:v2.1.1"
ROOK_CSI_ATTACHER_IMAGE: "quay.io/k8scsi/csi-attacher:v2.1.0"
ROOK_CSI_RESIZER_IMAGE: "quay.io/k8scsi/csi-resizer:v0.4.0"
```

Use default images

If you would like Rook to use the inbuilt default upstream images, then you may simply remove all variables matching `ROOK_CSI_*_IMAGE` from the above ConfigMap and/or the operator deployment.

Verifying updates

You can use the below command to see the CSI images currently being used in the cluster.

```
# kubectl --namespace rook-ceph get pod -o jsonpath='{range .items[*]}{range .spec.containers[*]}{.image}
{"\n"}' -l 'app in (csi-rbdplugin,csi-rbdplugin-provisioner,csi-cephfsplugin,csi-cephfsplugin-provisioner)' |
sort | uniq
quay.io/cephcsi/cephcsi:v3.1.0
```

```
quay.io/k8scsi/csi-attacher:v2.1.0
quay.io/k8scsi/csi-node-driver-registrar:v1.2.0
quay.io/k8scsi/csi-provisioner:v1.6.0
quay.io/k8scsi/csi-resizer:v0.4.0
quay.io/k8scsi/csi-snapshotter:v2.1.1
quay.io/k8scsi/csi-resizer:v0.4.0
```

Cleaning up a Cluster

If you want to tear down the cluster and bring up a new one, be aware of the following resources that will need to be cleaned up:

- `rook-ceph` namespace: The Rook operator and cluster created by `operator.yaml` and `cluster.yaml` (the cluster CRD)
- `/var/lib/rook`: Path on each host in the cluster where configuration is cached by the ceph mons and osds

Note that if you changed the default namespaces or paths such as `dataDirHostPath` in the sample yaml files, you will need to adjust these namespaces and paths throughout these instructions.

If you see issues tearing down the cluster, see the Troubleshooting section below.

If you are tearing down a cluster frequently for development purposes, it is instead recommended to use an environment such as Minikube that can easily be reset without worrying about any of these steps.

Delete the Block and File artifacts

First you will need to clean up the resources created on top of the Rook cluster.

These commands will clean up the resources from the block and file walkthroughs (unmount volumes, delete volume claims, etc). If you did not complete those parts of the walkthrough, you can skip these instructions:

```
kubectl delete -f ../wordpress.yaml
kubectl delete -f ../mysql.yaml
kubectl delete -n rook-ceph cephblockpool replicapool
kubectl delete storageclass rook-ceph-block
kubectl delete -f csi/cephfs/kube-registry.yaml
kubectl delete storageclass csi-cephfs
```

Delete the CephCluster CRD

After those block and file resources have been cleaned up, you can then delete your Rook cluster. This is important to delete **before removing the Rook operator and agent or else resources may not be cleaned up properly**.

```
kubectl -n rook-ceph delete cephcluster rook-ceph
```

Verify that the cluster CRD has been deleted before continuing to the next step.

```
kubectl -n rook-ceph get cephcluster
```

Delete the Operator and related Resources

This will begin the process of the Rook Ceph operator and all other resources being cleaned up. This includes related resources such as the agent and discover daemonsets with the following commands:

```
kubectl delete -f operator.yaml
kubectl delete -f common.yaml
```

Delete the data on hosts

> **IMPORTANT**: The final cleanup step requires deleting files on each host in the cluster. All files under the `dataDirHostPath` property specified in the cluster CRD will need to be deleted. Otherwise, inconsistent state will remain when a new cluster is started.

Connect to each machine and delete `/var/lib/rook`, or the path specified by the `dataDirHostPath`.

In the future this step will not be necessary when we build on the K8s local storage feature.

If you modified the demo settings, additional cleanup is up to you for devices, host paths, etc.

Zapping Devices

Disks on nodes used by Rook for osds can be reset to a usable state with the following methods:

```
#!/usr/bin/env bash
DISK="/dev/sdb"
# Zap the disk to a fresh, usable state (zap-all is important, b/c MBR has to be clean)
# You will have to run this step for all disks.
sgdisk --zap-all $DISK
# Clean hdds with dd
dd if=/dev/zero of="$DISK" bs=1M count=100 oflag=direct,dsync
# Clean disks such as ssd with blkdiscard instead of dd
blkdiscard $DISK

# These steps only have to be run once on each node
# If rook sets up osds using ceph-volume, teardown leaves some devices mapped that lock the disks.
ls /dev/mapper/ceph-* | xargs -I% -- dmsetup remove %
# ceph-volume setup can leave ceph-<UUID> directories in /dev (unnecessary clutter)
rm -rf /dev/ceph-*
```

Troubleshooting

If the cleanup instructions are not executed in the order above, or you otherwise have difficulty cleaning up the cluster, here are a few things to try.

The most common issue cleaning up the cluster is that the `rook-ceph` namespace or the cluster CRD remain indefinitely in the `terminating` state. A namespace cannot be removed until all of its resources are removed, so look at which resources are pending termination.

Look at the pods:

```
1. kubectl -n rook-ceph get pod
```

If a pod is still terminating, you will need to wait or else attempt to forcefully terminate it (`kubectl delete pod <name>`).

Now look at the cluster CRD:

```
1. kubectl -n rook-ceph get cephcluster
```

If the cluster CRD still exists even though you have executed the delete command earlier, see the next section on removing the finalizer.

Removing the Cluster CRD Finalizer

When a Cluster CRD is created, a finalizer is added automatically by the Rook operator. The finalizer will allow the operator to ensure that before the cluster CRD is deleted, all block and file mounts will be cleaned up. Without proper cleanup, pods consuming the storage will be hung indefinitely until a system reboot.

The operator is responsible for removing the finalizer after the mounts have been cleaned up. If for some reason the operator is not able to remove the finalizer (ie. the operator is not running anymore), you can delete the finalizer manually with the following command:

```
    for CRD in $(kubectl get crd -n rook-ceph | awk '/ceph.rook.io/ {print $1}'); do kubectl patch crd -n rook-
1. ceph $CRD --type merge -p '{"metadata":{"finalizers": [null]}}'; done
```

This command will patch the following CRDs on v1.3:

```
1. cephblockpools.ceph.rook.io
2. cephclients.ceph.rook.io
3. cephfilesystems.ceph.rook.io
4. cephnfses.ceph.rook.io
5. cephobjectstores.ceph.rook.io
6. cephobjectstoreusers.ceph.rook.io
```

Within a few seconds you should see that the cluster CRD has been deleted and will no longer block other cleanup such as deleting the `rook-ceph` namespace.

EdgeFS Data Fabric

EdgeFS is high-performance and fault-tolerant decentralized data fabric with virtualized access to S3 object, NFS/SMB file, NoSQL and iSCSI block.

EdgeFS is capable of spanning unlimited number of geographically distributed sites (Geo-site), connected with each other as one global name space data fabric running on top of Kubernetes platform, providing persistent, fault-tolerant and high-performance volumes for stateful Kubernetes Applications.

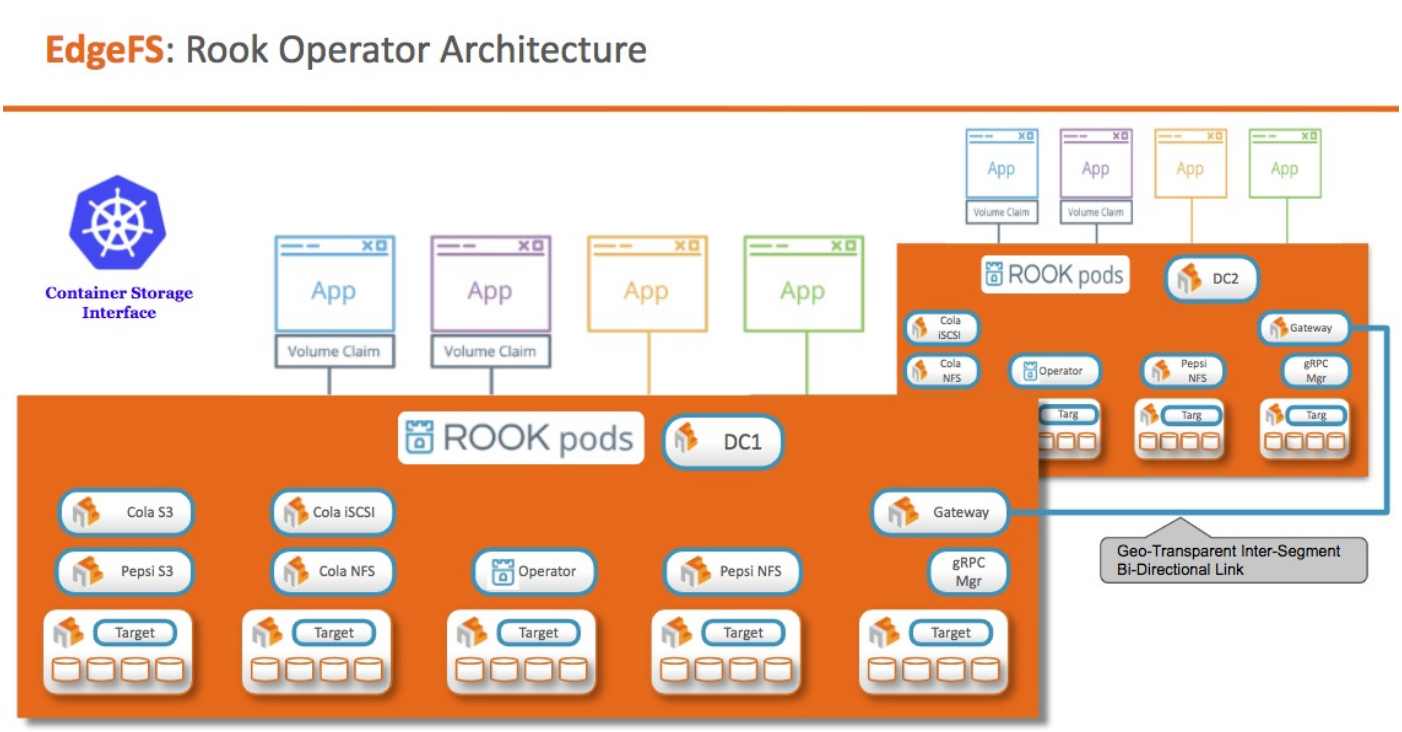
At each Geo-site, EdgeFS nodes deployed as containers (StatefulSet) on physical or virtual Kubernetes nodes, pooling available storage capacity and presenting it via compatible S3/NFS/SMB/iSCSI/etc storage emulated protocols for cloud-native applications running on the same or dedicated servers.

How it works, in a Nutshell?

If you familiar with “git”, where all modifications are fully versioned and globally immutable, it is highly likely you already know how it works at its core. Think of it as a world-scale copy-on-write technique. Now, if we can make a parallel for you to understand it better - what EdgeFS does, it expands “git” paradigm to object storage and making Kubernetes Persistent Volumes accessible via emulated storage standard protocols e.g. S3, NFS, SMB and even block devices such as iSCSI, in a high-performance and low-latency ways. With fully versioned modifications, fully immutable metadata and data, users data can be transparently replicated, distributed and dynamically pre-fetched across many Geo-sites.

Design

Rook enables easy deployment of EdgeFS Geo-sites on Kubernetes using Kubernetes primitives.



With Rook running in the Kubernetes cluster, Kubernetes PODs or External applications can mount block devices and filesystems managed by Rook, or can use the S3/S3X API for object storage. The Rook operator automates configuration of storage components and monitors the cluster to ensure the storage remains available and healthy.

The Rook operator is a simple container that has all that is needed to bootstrap and monitor the storage cluster. The operator will start and monitor StatefulSet storage Targets, gRPC manager and Prometheus Multi-Tenant Dashboard. All the attached devices (or directories) will provide pooled storage site. Storage sites then can be easily connected with each other as one global name space data fabric. The operator manages CRDs for Targets, Scale-out NFS/SMB, Object stores (S3/S3X), and iSCSI volumes by initializing the pods and other artifacts necessary to run the services.

The operator will monitor the storage Targets to ensure the cluster is healthy. EdgeFS will dynamically handle services failover, and other adjustments that maybe made as the cluster grows or shrinks.

The EdgeFS Rook operator also comes with tightly integrated CSI plugin. CSI pods deployed on every Kubernetes node. All storage operations required on the node are handled such as attaching network storage devices, mounting NFS exports, and dynamic provisioning.

Rook is implemented in golang. EdgeFS is implemented in Go/C where the data path is highly optimized.

Learn more at edgefs.io.

EdgeFS Cluster CRD

Rook allows creation and customization of storage clusters through the custom resource definitions (CRDs).

Sample

To get you started, here is a simple example of a CRD to configure a EdgeFS cluster with just one local per-host directory /data:

```
apiVersion: edgefs.rook.io/v1
kind: Cluster
metadata:
  name: rook-edgefs
  namespace: rook-edgefs
spec:
  edgefsImageName: edgefs/edgefs:latest
  serviceAccount: rook-edgefs-cluster
  dataDirHostPath: /data
  storage:
    useAllNodes: true   # use only for test deployments
```

or if you have raw block devices provisioned, it can dynamically detect, format and utilize all raw devices on all nodes with simple CRD as below:

```
apiVersion: edgefs.rook.io/v1
kind: Cluster
metadata:
  name: rook-edgefs
  namespace: rook-edgefs
spec:
  edgefsImageName: edgefs/edgefs:latest
  serviceAccount: rook-edgefs-cluster
  dataDirHostPath: /data
  storage:
    useAllNodes: true   # use only for test deployments
    useAllDevices: true
```

or if you want to just install it on **single node1, single SSD device /dev/sdb**, use this sample:

```
apiVersion: edgefs.rook.io/v1
kind: Cluster
metadata:
  name: rook-edgefs
  namespace: rook-edgefs
spec:
```

```
  edgefsImageName: edgefs/edgefs:latest
  serviceAccount: rook-edgefs-cluster
  dataDirHostPath: /data
  sysRepCount: 1
  failureDomain: "device"
  storage:
    useAllNodes: false
    useAllDevices: false
    config:
      useAllSSD: "true"
      useMetadataOffload: "false"
    nodes:
    - name: "node1"
      devices:
      - name: "sdb"
```

or if you want to just install it on **single node1, single directory /media**, use this sample:

```
apiVersion: edgefs.rook.io/v1
kind: Cluster
metadata:
  name: rook-edgefs
  namespace: rook-edgefs
spec:
  edgefsImageName: edgefs/edgefs:latest
  serviceAccount: rook-edgefs-cluster
  dataDirHostPath: /data
  sysRepCount: 1
  failureDomain: "device"
  storage:
    useAllNodes: false
    useAllDevices: false
    directories:
    - path: /media   # global for all nodes, cannot be per-node!
    nodes:
    - name: "node1"
```

In addition to the CRD, you will also need to create a namespace, role, and role binding as seen in the common cluster resources below.

Settings

Settings can be specified at the global level to apply to the cluster as a whole, while other settings can be specified at more fine-grained levels, e.g. individual nodes. If any setting is unspecified, a suitable default will be used automatically.

Cluster metadata

- `name`: The name that will be used internally for the EdgeFS cluster. Most commonly the name is the same as the namespace since multiple clusters are not supported in the same namespace.
- `namespace`: The Kubernetes namespace that will be created for the Rook cluster. The services, pods, and other resources created by the operator will be added to this namespace. The common scenario is to create a single Rook cluster. If multiple clusters are created, they must not have conflicting devices or host paths.
- `edgefsImageName`: EdgeFS image to use. If not specified then `edgefs/edgefs:latest` is used. We recommend to specify particular image version for production use, for example `edgefs/edgefs:1.2.124`.

Cluster Settings

- `dataDirHostPath`: The path on the host (hostPath) where config and data should be stored for each of the services. If the directory does not exist, it will be created. Because this directory persists on the host, it will remain after pods are deleted. If `storage` settings not provided then provisioned hostPath will also be used as a storage device for Target pods (automatic provisioning via `rtlfs`).
  - On **Minikube** environments, use `/data/rook`. Minikube boots into a tmpfs but it provides some directories where files can be persisted across reboots. Using one of these directories will ensure that Rook's data and configuration files are persisted and that enough storage space is available.
  - **WARNING**: For test scenarios, if you delete a cluster and start a new cluster on the same hosts, the path used by `dataDirHostPath` must be deleted. Otherwise, stale information and other config will remain from the previous cluster and the new target will fail to start. If this value is empty, each pod will get an ephemeral directory to store their config files that is tied to the lifetime of the pod running on that node. More details can be found in the Kubernetes empty dir docs.
- `dataVolumeSize`: Alternative to `dataDirHostPath`. If defined then Cluster CRD operator will disregard `dataDirHostPath` setting and instead will automatically claim persistent volume. If `storage` settings not provided then provisioned volume will also be used as a storage device for Target pods (automatic provisioning via `rtlfs`).
- `sysRepCount`: overrides the default (3) system replication count value. Can be set to 1 or 2 for a cluster with limited number of failure domains. For example, a signle node setup with two disks can provide up to 2 replicas per chunk and requires `sysRepCount` to be set to 1 or 2.
- `failureDomain`: identifies the way chunk replicas are distributed accross cluster's disks. The `device` domain requires each replicas to resides on different disks, the `host` domains implies one replica per node and the `zone` domain is for one replica per zone. If `failureDomain` option isn't specified, then the failure domain is set to `host` or `zone` depending on nodes config (see below). The `failureDomain` allows to specify the failure domain explicitly. For example, a single-node cluster with multiple nodes requires the `failureDomain` set to `device` if the `sysRepCount` > 1.

- `dashboard`: This specification may be used to override and enable additional EdgeFS UI Dashboard functionality.
  - `localAddr`: Specifies local IP address to be used as Kubernetes external IP.
- `network`: network configuration settings
- `devicesResurrectMode`: When enabled, this mode attempts to recreate cluster based on previous CRD definition. If this flag set to one of the parameters, then operator will only adjust networking. Often used when clean up of old devices is needed. Only applicable when used with `dataDirHostPath`.
  - `restore`: Attempt to restart and restore previously enabled cluster CRD.
  - `restoreZap`: Attempt to re-initialize previously selected `devices` prior to restore. By default cluster assumes that selected devices have no logical partitions and considered empty.
  - `restoreZapWait`: Attempt to cleanup previously selected `devices` and wait for cluster delete. This is useful when clean up of old devices is needed. Additional containers count should be specified if cluster was originally created with a total per-node capacity that exceeding `maxContainerCapacity` option, e.g., `devicesResurrectMode: "restoreZapWait: 2"`.
- `serviceAccount`: The service account under which the EdgeFS pods will run that will give access to ConfigMaps in the cluster's namespace. If not set, the default of `rook-edgefs-cluster` will be used.
- `chunkCacheSize`: Limit amount of memory allocated for dynamic chunk cache. By default Target pod uses up to 75% of available memory as chunk caching area. This option can influence this allocation strategy.
- `placement`: placement configuration settings
- `resourceProfile`: Cluster segment wide resource utilization profile (Memory and CPU). Can be `embedded` or `performance` (default). In case of `performance` each Target pod requires at least 8Gi of memory and 4 CPU cores in terms of to operate efficiently. If `resources` limits are set to less then 8Gi of memory then operator will automatically set profile to `embedded`. In `embedded` profile case, Target pod requires 1Gi of memory and 2 CPU cores, where memory allocation is split between number of PLevels (see `rtPLevelOverride` option) with 64Mi minimally per one PLevel, 64Mi for Target pod itself and the rest for chunk cache (see `chunkCacheSize` option) that allocates up to 75% of available memory.
- `resources`: resources configuration settings
- `storage`: Storage selection and configuration that will be used across the cluster. Note that these settings can be overridden for specific nodes.
  - `useAllNodes`: `true` or `false`, indicating if all nodes in the cluster should be used for storage according to the cluster level storage selection and configuration values. If individual nodes are specified under the `nodes` field below, then `useAllNodes` must be set to `false`.
  - `nodes`: Names of individual nodes in the cluster that should have their storage included in accordance with either the cluster level configuration specified above or any node specific overrides described in the next section below. `useAllNodes` must be set to `false` to use specific nodes and their config.
  - storage selection settings

    - storage configuration settings
- `skipHostPrepare`: By default all nodes selected for EdgeFS deployment will be automatically configured via preparation jobs. If this option set to `true` node configuration will be skipped.
- `trlogProcessingInterval`: Controls for how many seconds cluster would aggregate object modifications prior to processing it by accounting, bucket updates, ISGW Links and notifications components. Has to be defined in seconds and must be composite of 60, i.e. 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30. Default is 10. Recommended range is 2 - 20. This is cluster wide setting and cannot be easily changed after cluster is created. Any new node added has to reflect exactly the same setting.
- `trlogKeepDays`: Controls for how many days cluster need to keep transaction log interval batches with version manifest references. If you planning to have cluster disconnected from ISGW downlinks for longer period time, consider to increase this value. Default is 3. This is cluster wide setting and cannot be easily changed after cluster is created. The value can be fractional, e.g. 1.1 or 0.1
- `maxContainerCapacity`: Overrides default total disks capacity per target container. Default is “132Ti”.
- `useHostLocalTime`: Force usage of the host’s /etc/localtime inside EdgeFS containers. Default is `false`.

Node Updates

Nodes can be added and removed over time by updating the Cluster CRD, for example with `kubectl -n rook-edgefs edit cluster.edgefs.rook.io rook-edgefs`. This will bring up your default text editor and allow you to add and remove storage nodes from the cluster. This feature is only available when `useAllNodes` has been set to `false`.

Node Settings

In addition to the cluster level settings specified above, each individual node can also specify configuration to override the cluster level settings and defaults. If a node does not specify any configuration then it will inherit the cluster level settings.

- `name`: The name of the node, which should match its `kubernetes.io/hostname` label.
- `config`: Config settings applied to all VDEVs on the node unless overridden by `devices` or `directories`. See the config settings below.
- storage selection settings
- storage configuration settings

Storage Selection Settings

Below are the settings available, both at the cluster and individual node level, for selecting which storage resources will be included in the cluster.

- `useAllDevices`: `true` or `false`, indicating whether all devices found on nodes in the

cluster should be automatically consumed by Targets. This is recommended for controlled environments where you will not risk formatting of devices with existing data. When `true`, all devices will be used except those with partitions created or a local filesystem. This can be overridden by `deviceFilter`. **warning** Don't set this option to `true` for RTKVS disk backend.

- `deviceFilter`: A regular expression that allows selection of devices to be consumed by target. If individual devices have been specified for a node then this filter will be ignored. This field uses golang regular expression syntax. For example:
  - `sdb`: Only selects the `sdb` device if found
  - `^sd.`: Selects all devices starting with `sd`
  - `^sd[a-d]`: Selects devices starting with `sda`, `sdb`, `sdc`, and `sdd` if found
  - `^s`: Selects all devices that start with `s`
  - `^[^r]`: Selects all devices that do *not* start with `r`
- `devices`: A list of individual device names belonging to this node to include in the storage cluster. Mixing of `devices` and `directories` on the same node isn't supported.
  - `name`: The name of the device (e.g., `sda`).
  - `fullpath`: The full path to the device (e.g., `/dev/disk/by-id/scsi-35000c5008335c83f`). If specified then `name` can be omitted.
  - `config`: Device-specific config settings. See the config settings below.
- `directories`: A list of directory paths on the nodes that will be included in the storage cluster. Note that using two directories on the same physical device can cause a negative performance impact. Since EdgeFS is leveraging StatefulSet, directories can only be defined at cluster level. Mixing of `devices` and `directories` on the same node isn't supported unless `rtkvs` disk engine is used. ** note **: For the `rtkvs` disk engine, at least one directory needs to be provided in order to store some small amount of metadata. For performance reasons, it would be better to have one directory per disk.
  - `path`: The path on disk of the directory (e.g., `/rook/storage-dir`).
  - `config`: Directory-specific config settings. See the config settings below.

Storage Configuration Settings

The following storage selection settings are specific to EdgeFS and do not apply to other backends. All variables are key-value pairs represented as strings. While EdgeFS supports multiple backends, it is not recommended to mix them within same cluster. In case of `devices` (physical or emulated raw disks), EdgeFS will automatically use `rtrd` backend unless `useRtkvsBackend` is specified. In the latter case, the `rtkvs` engine will be chosen. In all other cases `rtlfs` (local filesystem) will be used.

> **IMPORTANT**: Keys needs to be case-sensitive and values has to be provided as strings.

- `useRtkvsBackend`: forces the cluster use the `rtkvs` disk engine and setting's value selects a key-value backend to be used. At the moment there is only backend named the `kvssd` for Samsung's KV SSD. The usage of `rtkvs` engine implies definition of one or several `device.name` or `device.fullpath` settings which have to point to

backend's disk entries (see cluster_kvsdd.yaml).

- `walMode` : allows to enable/disable the Write-Ahead log (WAL). For `rtlfs` and `rtkvs` there are two options: `on` to enable or `off` to disable. It's better to keep it `on` unless `useAllSSD` or `useRtkvsBackend` are used. For `rtkvs`, there is an extra option: `metadata` which implies usage of WAL for data types which aren't stored on `rtkvs` backend (a KVSSD).
- `useMetadataOffload` : Dynamically detect appropriate SSD/NVMe device to use for the metadata on each node. Performance can be improved by using a low latency device as the metadata device, while other spinning platter (HDD) devices on a node are used to store data. Typical and recommended proportion is in range of 1:1 - 1:6. Default is false. Applicable only to `rtrd` .
- `useMetadataMask` : Defines what parts of metadata needs to be stored on offloaded devices. Default is 0xff, offload all metadata. To save SSD/NVMe capacity, set it to 0x7d to offload all except second level manifests. Applicable only to `rtrd` .
- `useBCache` : When `useMetadataOffload` is true, enable use of BCache. Default is false. Applicable only to `rtrd` and when host has "bcache" kernel module preloaded.
- `useBCacheWB` : When `useMetadataOffload` and `useBCache` is true, this option can enable use of BCache write-back cache. By default BCache only used as read cache in front of HDD. Applicable only to `rtrd` .
- `useAllSSD` : When set to true, only SSD/NVMe non rotational devices will be used. Default is false and if `useMetadataOffload` not defined then only rotational devices (HDDs) will be picked up during node provisioning phase. Is not applicabel to `rtkvs` .
- `rtPLevelOverride` : In case of large devices or directories, it will be automatically partitioned into smaller parts around 500GB each. In case of embedded use cases, lowering the value would allow to operate with smaller memory footprint devices at the cost of performance. This option allows partitioning number override. Default is automatic. Typical and recommended range is 1 - 32. For `rtkvs` recomended plevel is 16.
- `hddReadAhead` : For all HDD or hybrid (SSD/HDD) use cases, adjusting hddReadAhead may provide significant boost in performance. Set to a value higher then 0, in KBs. Not applicable to `rtkvs`
- `mdReserved` : For hybrid (SSD/HDD) use case, adjusting mdReserved can be necessary when combined with BCache read/write caches. Allowed range 10-99% of automatically calcuated slice. Not applicable to `rtkvs`
- `rtVerifyChid` : Verify transferred or read payload. Payload can be data or metadata chunk of flexible size between 4K and 8MB. EdgeFS uses SHA-3 variant to cryptographically sign each chunk and uses it for self validation, self healing and FlexHash addressing. In case of low CPU systems verification after networking transfer prior to write can be disabled by setting this parameter to 0. In case of high CPU systems, verification after read but before networking transfer can be enabled by setting this parameter to 2. Default is 1, i.e. verify after networking transfer only. Setting it to 0 may improve CPU utilization at the cost of reduced availability. However, for objects with 3 or more replicas, availability isn't going to be visibly affected.
- `lmdbPageSize` : Defines default LMDB page size in bytes. Default is 16384. For

capacity (all HDD) or hybrid (HDD/SSD) systems consider to increase this value to 32768 to achieve higher throughput performance. For all SSD and small database workloads, consider to decrease this to 8192 to achieve lower latency and higher IOPS. Please be advised that smaller values MAY cause fragmentation. Acceptable values are 4096, 8192, 16384 and 32768. Not applicable to `rtkvs`

- `lmdbMdPageSize`: Defines SSD metadata offload LMDB page size in bytes. Default is 8192. For large amount of small objects or files, consider to decrease this to 4096 to achieve better SSD capacity utilization. Acceptable values are 4096, 8192, 16384 and 32768. Not applicable to `rtkvs`
- `sync`: Defines default behavior of write operations at device or directory level. Acceptable values are 0, 1 (default), 2, 3.
  - `0`: No syncing will happen. Highest performance possible and good for HPC scratch types of deployments. This option will still sustain crash of pods or software bugs. It will not sustain server power loss an may cause node / device level inconsistency.
  - `1`: Default method. Will guarantee node / device consistency in case of power loss with reduced durability.
  - `2`: Provides better durability in case of power loss at the cost of extra metadata syncing.
  - `3`: Most durable and reliable option at the cost of significant performance impact.
- `maxSizeGB`: For `rtlfs`, defines maximum allowed size to use per directory in gigabytes. For `rtkvs` this is the maximum space the disk's metadata table can occupy.
- `zone`: Enables the node's failure domain number. Default value is 0 (no zoning). Zoning number is a logical failure domain tagging mechanism and if enabled then it has to be set for all the nodes in the cluster. See also, the `failureDomain`
- `noIP4Frag`: When set to `true` it prevents sending fragmented UDP traffic. **IMPORTANT**: maximum data chunk size will be redused significantly.
- `sysMaxChunkSize`: Set maximum allowed data chunk size expressed in bytes. Must be a power of two value. Default is 1M.
- `payloadS3URL`: When set, it activates payload data chunk forwarding to an external S3 server. The value is an URL of the S3 bucket the payload chunks will be put to. Example: http://s3.aws-region.amazonaws.com/bucket. Only applicable for RTRD (raw disk) engine. Disabled by default.
- `payloadS3Region`: S3 server region. Default is `us-east-1`. Only applicable when the `payloadS3URL` is set.
- `payloadS3MinKb`: a minimal payload chunk size to trigger the forwarding to the S3 bucket. Data chunks smaller than this value will be stored locally. Only applicable when the `payloadS3URL` is set.
- `payloadS3CapacityGB`: capacity of the external S3 bucket expressed in GB. This is an artificial value used to report accurate usage summary and to limit storage size. Only applicable when the `payloadS3URL` is set.
- `payloadS3Secret`: name of a Kubernetes secret to be used as an external S3 bucket credential. The secret needs to be pre-creted before deployment of the EdgeFS cluster and resides in the same namespace as the cluster. Secret's key neeed to

be set to `cred` and value format is as follow: `<aws_key>,<aws_secret>`. See an example in the s3PayloadSecret.yaml

Placement Configuration Settings

Placement configuration for the cluster services. It includes the following keys: `mgr`, `target` and `all`. Each service will have its placement configuration generated by merging the generic configuration under `all` with the most specific one (which will override any attributes).

A Placement configuration is specified (according to the Kubernetes PodSpec) as:

- `nodeAffinity`: Kubernetes NodeAffinity
- `podAffinity`: Kubernetes PodAffinity
- `podAntiAffinity`: Kubernetes PodAntiAffinity
- `tolerations`: list of Kubernetes Toleration

The `mgr` pod does not allow `Pod` affinity or anti-affinity. This is because of the mgrs having built-in anti-affinity with each other through the operator. The operator chooses which nodes are to run a mgr on. Each mgr is then tied to a node with a node selector using a hostname.

Network Configuration Settings

Configure the network that will be enabled for the cluster and services. This is optional and if not defined then the cluster default network's `eth0` will be used to construct cluster bucket network.

- `provider`: Specifies the network provider that will be used to connect the network interface. You can choose between `host`, and `multus`.
- `selectors`: List the network selector that will be used associated by a key. The available keys are `server` and `broker`.
    - `server`: Specifies data daemon host's networking interface name or multus's network attachment selection annotation.
    - `broker`: Specifies broker daemon host's networking interface name or multus's network attachment selection annotation.

For `multus` network provider, an already working cluster with multus networking is required. Network attachment definition that later will be attached to the cluster needs to be created before the Cluster CRD. You can add the multus network attachment selection annotation selecting the created network attachment definition on `selectors`. Make sure to define the interface name that will be assigned by multus and choose only one syntax either the short or JSON form to define all the available keys.

Cluster-wide Resources Configuration Settings

Resources should be specified so that the rook components are handled after Kubernetes

Pod Quality of Service classes. This allows to keep rook components running when for example a node runs out of memory and the rook components are not killed depending on their Quality of Service class.

You can set resource requests/limits for rook components through the Resource Requirements/Limits structure in the following keys:

- `mgr`: Set resource requests/limits for Mgrs.
- `target`: Set resource requests/limits for Targets.

Resource Requirements/Limits

For more information on resource requests/limits see the official Kubernetes documentation: Kubernetes - Managing Compute Resources for Containers.

- `requests`: Requests for cpu or memory.
  - `cpu`: Request for CPU (example: one CPU core `1`, 50% of one CPU core `500m`).
  - `memory`: Limit for Memory (example: one gigabyte of memory `1Gi`, half a gigabyte of memory `512Mi`).
- `limits`: Limits for cpu or memory.
  - `cpu`: Limit for CPU (example: one CPU core `1`, 50% of one CPU core `500m`).
  - `memory`: Limit for Memory (example: one gigabyte of memory `1Gi`, half a gigabyte of memory `512Mi`).

Kubernetes node labeling and selection

By default each Kubernetes node, available to deploy EdgeFS over it, will be treated as “target” EdgeFS instance. But cluster administrator able to label node as “gateway” node, such node will have no devices prepared for EdgeFS and will be used as EdgeFS dedicated service node. To mark a node as a “gateway”, the administrator can add a specific label to a node. Label format: `<edgefs-namespace>-nodetype=gateway`, where `edgefs-namespace` is current namespace for EdgeFS cluster deployment, by default is `rook-edgefs`. Example: `kubectl label node "k8s node name" rook-edgefs-nodetype=gateway`

Samples

Here are several samples for configuring EdgeFS clusters. Each of the samples must also include the namespace and corresponding access granted for management by the EdgeFS operator. See the common cluster resources below.

Storage configuration: All devices, All SSD/NVMes.

```
apiVersion: edgefs.rook.io/v1
kind: Cluster
metadata:
  name: rook-edgefs
```

```
  namespace: rook-edgefs
spec:
  edgefsImageName: edgefs/edgefs:latest
  dataDirHostPath: /var/lib/rook
  serviceAccount: rook-edgefs-cluster
  # cluster level storage configuration and selection
  storage:
    useAllNodes: true
    useAllDevices: true
    deviceFilter:
    location:
    config:
      useAllSSD: true
```

Storage Configuration: Specific devices

Individual nodes and their config can be specified so that only the named nodes below will be used as storage resources. Each node's 'name' field should match their 'kubernetes.io/hostname' label.

```
apiVersion: edgefs.rook.io/v1
kind: Cluster
metadata:
  name: rook-edgefs
  namespace: rook-edgefs
spec:
  edgefsImageName: edgefs/edgefs:latest
  dataDirHostPath: /var/lib/rook
  serviceAccount: rook-edgefs-cluster
  # cluster level storage configuration and selection
  storage:
    useAllNodes: false
    useAllDevices: false
    deviceFilter:
    location:
    config:
      rtVerifyChid: "0"
    nodes:
    - name: "172.17.4.201"
      devices:             # specific devices to use for storage can be specified for each node
      - name: "sdb"
      - name: "sdc"
      config:         # configuration can be specified at the node level which overrides the cluster level
config
        rtPLevelOverride: 8
    - name: "172.17.4.301"
      deviceFilter: "^sd."
```

Storage Configuration: Samsung's KV SSD

A single nodes configuration with 2 KV SSDs. The host's /media directory should have at least 64GB of free space for metadata.

```
spec:
  edgefsImageName: edgefs/edgefs:1.2.0
  serviceAccount: rook-edgefs-cluster
  dataDirHostPath: /var/lib/edgefs
  sysRepCount: 2
  failureDomain: "device"
  storage:
    useAllNodes: false
    directories:
    - path: /media
    useAllDevices: false
    config:
      useRtkvsBackend: kvssd
      rtPLevelOverride: "16"
      maxSizeGB: "32"
      sync: "0"
      walMode: "off"
    nodes:
    - name: "node1"
      devices:
      - fullpath: "/dev/disk/by-id/nvme-SAMSUNG_MZQLB3T8HALS-000AZ_S3VJNY0J600450"
      - fullpath: "/dev/disk/by-id/nvme-SAMSUNG_MZQLB3T8HALS-000AZ_S3VJNY0K303383"
```

Storage Configuration: payload forwarding to S3

A 4-nodes configuration with S3 payload forwarding enabled. Follow the next steps:

1. Create an EdgeFS namespace, default name is "rook-edgefs": `kubectl create ns rook-edgefs`
2. Create a S3 payload secrets in the EdgeFS namespace:

```
apiVersion: v1
kind: Secret
metadata:
  name: node185-s3payload-secret
  namespace: rook-edgefs
type: Opaque
data:
  cred: bm9kZTEsbm9kZTEK  # aws key/secret made by a command "echo node1,node1 | base64"
---
apiVersion: v1
kind: Secret
metadata:
  name: node186-s3payload-secret
  namespace: rook-edgefs
type: Opaque
data:
  cred: bm9kZTIsbm9kZTIK
---
```

```
apiVersion: v1
kind: Secret
metadata:
  name: cluster-s3payload-secret
  namespace: rook-edgefs
type: Opaque
data:
  cred: Y2x1c3RlcixjbHVzdGVyCg==
```

1. Create EdgeFS cluster:

```
spec:
  edgefsImageName: edgefs/edgefs:1.2.0
  serviceAccount: rook-edgefs-cluster
  dataDirHostPath: /var/lib/edgefs
  storage:
    useAllNodes: false
    useAllDevices: false
    config:
      useAllSSD: "true"  # allSSD is a preferable configuration
      useMetadataOffload: "false"  # useMetadataOffload is not allowed
      payloadS3URL: "http://http://s3.us-west-1.amazonaws.com/bucket  # Default S3 bucket for all nodes
      payloadS3Region: "us-west-1"
      payloadS3MinKb: "128" # Payloads larger than 128K will got to S3 bucket
      payloadS3CapacityGB: "1024" # S3 bucket capacity is 1TB
      payloadS3Secret: "cluster-s3payload-secret" # Secter for default S3 bucket
    nodes:
    - name: node183 # node level storage configuration
      devices:
      - name: "sdb"
      - name: "sdc"
    - name: node184 # node level storage configuration
      devices: # specific devices to use for storage can be specified for each node
      - name: "sdb"
      - name: "sdc"
    - name: node185 # node level storage configuration
      devices: # specific devices to use for storage can be specified for each node
      - name: "sdb"
      - name: "sdc"
      config: # configuration can be specified at the node level which overrides the cluster level config
        payloadS3URL: "http://s3.asia.amazonaws.com/bucket185"
        payloadS3Region: "asia"
        payloadS3Capacity: "1099511627776"
        payloadS3Secret: "node185-s3payload-secret"
    - name: node186 # node level storage configuration
      devices: # specific devices to use for storage can be specified for each node
      - name: "sdb"
      - name: "sdc"
      config: # configuration can be specified at the node level which overrides the cluster level config
        payloadS3URL: "http://s3.asia.amazonaws.com/bucket186"
        payloadS3Region: "asia"
        payloadS3Capacity: "1099511627776"
```

```
        payloadS3Secret: "node186-s3payload-secret"
```

Node Affinity

To control where various services will be scheduled by Kubernetes, use the placement configuration sections below. The example under ‘all’ would have all services scheduled on Kubernetes nodes labeled with ‘role=storage’ and tolerate taints with a key of ‘storage-node’.

```
apiVersion: edgefs.rook.io/v1
kind: Cluster
metadata:
  name: rook-edgefs
  namespace: rook-edgefs
spec:
  edgefsImageName: edgefs/edgefs:latest
  dataDirHostPath: /var/lib/rook
  serviceAccount: rook-edgefs-cluster
  placement:
    all:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: role
              operator: In
              values:
              - storage-node
      tolerations:
      - key: storage-node
        operator: Exists
    mgr:
      nodeAffinity:
      tolerations:
    target:
      nodeAffinity:
      tolerations:
```

Resource requests/Limits

To control how many resources the rook components can request/use, you can set requests and limits in Kubernetes for them. You can override these requests/limits for Targts per node when using `useAllNodes: false` in the `node` item in the `nodes` list.

```
apiVersion: edgefs.rook.io/v1
kind: Cluster
metadata:
  name: rook-edgefs
  namespace: rook-edgefs
```

```
spec:
  edgefsImageName: edgefs/edgefs:latest
  dataDirHostPath: /var/lib/rook
  serviceAccount: rook-edgefs-cluster
  # cluster level resource requests/limits configuration
  resources:
  storage:
    useAllNodes: false
    nodes:
    - name: "172.17.4.201"
      resources:
        limits:
          cpu: "2"
          memory: "4096Mi"
        requests:
          cpu: "2"
          memory: "4096Mi"
```

Network Configuration: Multus network

An example on how to configure the cluster network to use multus network. Here, a NetworkAttachmentDefinition named flannel on rook-edgefs namespace is assumed.

```
apiVersion: edgefs.rook.io/v1
kind: Cluster
metadata:
  name: rook-edgefs
  namespace: rook-edgefs
spec:
  edgefsImageName: edgefs/edgefs:latest
  dataDirHostPath: /var/lib/rook
  serviceAccount: rook-edgefs-cluster
  network:
    provider: multus
    selectors:
      server: flannel@net1
      broker: flannel@net2
```

Common Cluster Resources

Each EdgeFS cluster must be created in a namespace and also give access to the Rook operator to manage the cluster in the namespace. Creating the namespace and these controls must be added to each of the examples previously shown.

```
apiVersion: v1
kind: Namespace
metadata:
  name: rook-edgefs
---
```

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: rook-edgefs-cluster
  namespace: rook-edgefs
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: rook-edgefs-cluster
  namespace: rook-edgefs
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: [ "get", "list", "watch", "create", "update", "delete" ]
- apiGroups: [""]
  resources: ["pods"]
  verbs: [ "get", "list" ]
---
# Allow the operator to create resources in this cluster's namespace
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: rook-edgefs-cluster-mgmt
  namespace: rook-edgefs
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: rook-edgefs-cluster-mgmt
subjects:
- kind: ServiceAccount
  name: rook-edgefs-system
  namespace: rook-edgefs-system
---
# Allow the pods in this namespace to work with configmaps
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: rook-edgefs-cluster
  namespace: rook-edgefs
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: rook-edgefs-cluster
subjects:
- kind: ServiceAccount
  name: rook-edgefs-cluster
  namespace: rook-edgefs
---
apiVersion: apps/v1
kind: PodSecurityPolicy
metadata:
  name: privileged
```

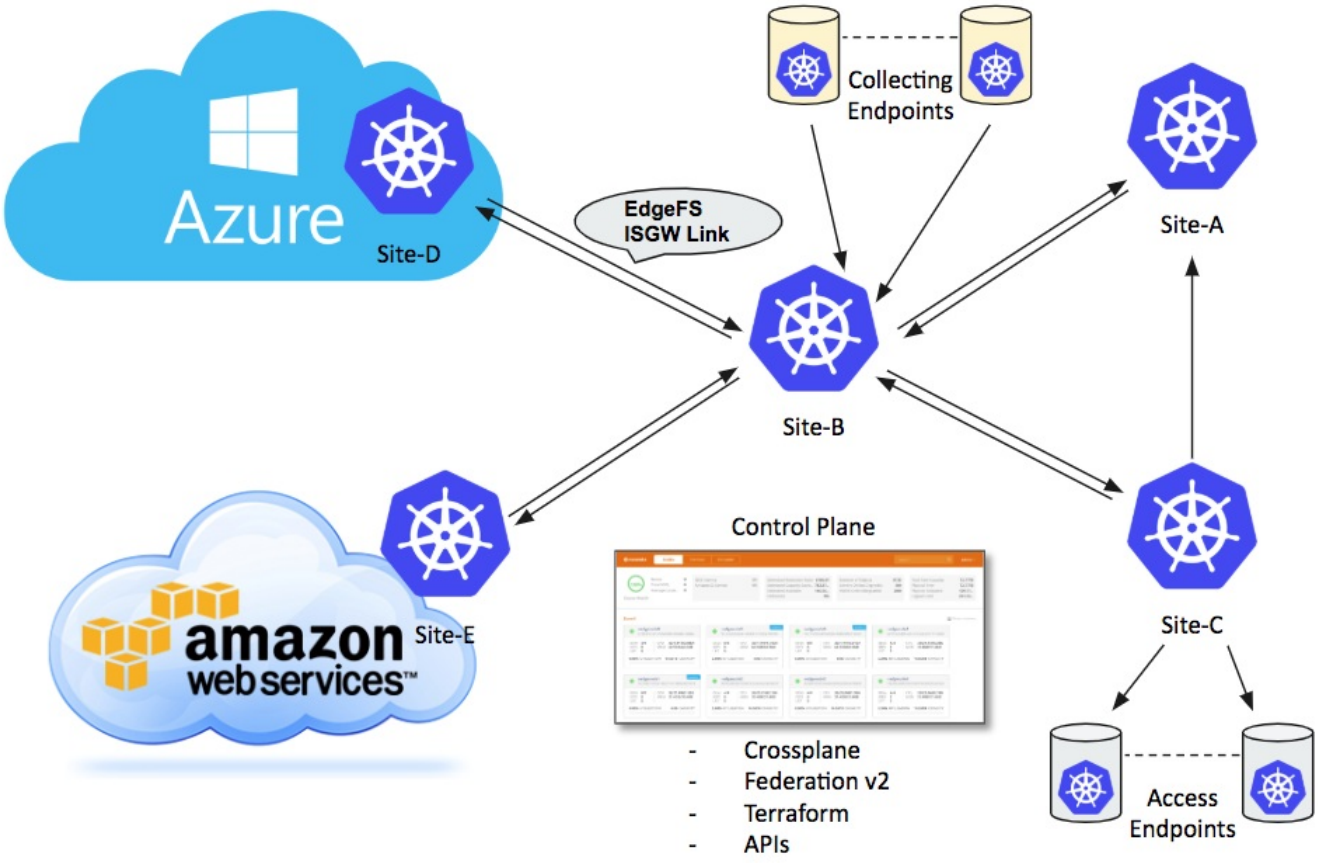
```yaml
spec:
  fsGroup:
    rule: RunAsAny
  privileged: true
  runAsUser:
    rule: RunAsAny
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  volumes:
  - '*'
  allowedCapabilities:
  - '*'
  hostPID: true
  hostIPC: true
  hostNetwork: false
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: privileged-psp-user
rules:
- apiGroups:
  - apps
  resources:
  - podsecuritypolicies
  resourceNames:
  - privileged
  verbs:
  - use
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: rook-edgefs-system-psp
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: privileged-psp-user
subjects:
- kind: ServiceAccount
  name: rook-edgefs-system
  namespace: rook-edgefs-system
```

EdgeFS ISGW (Inter-Segment GateWay) CRD

EdgeFS Inter-Segment Gateway link is a building block for EdgeFS cross-site, cross-cloud global namespace synchronization functionality.

It distributes modified chunks of data asynchronously and enables seamless as well as geographically transparent access to files, objects and block devices. It is important to note that a file or a block device consists of one or more objects, and so, within EdgeFS scope, ultimately everything is an object, globally immutable and self-validated.

To create an analogy, EdgeFS concept of global immutability of modifications very similar to how git operates with repository commits and branches. As such, this technique empowers EdgeFS users to construct and operate comprehensive wide-spread global namespaces with management overhead greatly simplified. A file or object modified at a source site where ISGW link is setup will be immediately noticed by ISGW endpoint links, thus spread out the change. Eventually, all the connected sites will receive file modification where only modified blocks get transferred.



Not only ISGW link reduces the amount of data needed to be transferred, it also deduplicating the transfers. Matching globally unique cryptographic signatures of a file change will not be transferred, thus enabling global namespace deduplication.

ISGW link can be bi-directional, i.e. enabling same file/object modifications across the namespaces. It works well for many use cases where application logic can ensure serialization of changes. Single bi-directional link can connect two sites but it is possible to create as many non-overlapping links as needed.

ISGW link can also transparently synchronize file, object, directory, bucket or tenant level snapshots, grouped into so-called SnapView construct. Thus, modification to a block device, for instance, can be consistently viewed across entire global namespace.

Because EdgeFS metadata is also globally immutable and unique, it is possible to enable a mode of transferring only metadata changes. With this mode enabled, users can construct efficient access endpoints where modifications can be fetched on demand, as a result creating globally and geographically distributed cache fog aggregation with a built-in E-LRU eviction policy.

Local I/O at each site executed with the speed of media devices used. Modifications transferred eventually, and not slowing down local site running application workflow.

EdgeFS Rook Operator allows creation and customization of High-Performance Inter-Segment synchronization links through the custom resource definitions (CRDs).

The following settings are available for customization of ISGW services.

Sample

```
apiVersion: edgefs.rook.io/v1
kind: ISGW
metadata:
  name: isgw01                              # Inter-Segment Gateway service name
  namespace: rook-edgefs
spec:
  direction: "send+receive"
  remoteURL: "ccow://192.168.1.43:14000"
  #replicationType: "initial"
  #metadataOnly: all
  #dynamicFetchAddr: 0.0.0.0:49600
  #localAddr: 0.0.0.0:10000
  #useEncryptedTunnel: true
  #chunkCacheSize: 1Gi
  #config:
  #   server: "isgws"
  #   clients: ["isgwc1", "isgwc2"]
  #placement:
  #  nodeAffinity:
  #    requiredDuringSchedulingIgnoredDuringExecution:
  #      nodeSelectorTerms:
  #      - matchExpressions:
  #        - key: rook-edgefs-nodetype
  #          operator: In
  #          values:
```

```
#          - isgw
#  tolerations:
#  - key: node230
#    operator: Exists
#  podAffinity:
#  podAntiAffinity:
#resourceProfile: embedded
resources:
#  limits:
#    cpu: "2"
#    memory: "4Gi"
#  requests:
#    cpu: "2"
#    memory: "4Gi"
# A key/value list of annotations
annotations:
#  key: value
```

Metadata

- `name`: The name of the ISGW link service to create, which must match existing EdgeFS service. Provided CRD metadata can override existing EdgeFS service metadata.
- `namespace`: The namespace of the Rook cluster where the ISGW service is created.
- `direction`: I/O flow direction: `send`, `receive` or `send+receive` (default).
- `remoteURL`: I/O flow with destination pointing to the other ISGW endpoint service, locally mounted directory or other (non-EdgeFS) S3 compatible service. Format should be of form PROTOCOL://LOCATION, where PROTOCOL can be of “ccow”, “file”, “s3” and LOCATION depends on the PROTOCOL. In case of “ccow” LOCATION needs to specify a HOST:PORT of the other EdgeFS site, e.g. ccow://192.168.1.43:14000. In case of “file” LOCATION needs to be a path to a local POSIX directory, e.g. file://volumeDir. In cae of “s3” LOCATION can either point to an existing bucket, e.g. s3://mybucket.aws.com or to the other S3 compatible server IP:HOST, e.g. s3://214.4.34.22:4567. It is important to note that file:// and s3:// protocols only provided for easy integration with existing infrastructure.
- `replicationType`: Defines how tenant’s buckets needs to be replicated. Can be specified as `initial`, `continuous` or `initial+continuous` (default). In case of `initial` replication type, ISGW service will only sync served buckets once with ability to manually trigger re-sync again. In case of `continuous` replication type, ISGW service will enable asynchronous syncing from the moment a bucket added to a served list of buckets, so that only recent changes will be synchronized, continuously. In case of `initial+continuous` replication type, ISGW service will first initiate syncing of all existing served buckets contents and once all is copied, it will automatically enable `continuous` mode of operation, guaranteeing that none of the source objects are skipped.
- `metadataOnly`: If set, then only immutable metadata changes will be transferred. This also enables on-demand data fetch and E-LRU cache capability. In case of

metadata only transfers, dynamically fetched data will be cached at the accessing endpoints ISGW links locations. If an application would want to read an offset of a data which isn't found, global search request will be formed and distributed across all the connected ISGW bi-directional links serving source bucket. Missing data chunks will be found and fetched, delivered transparently to application. Can be set to either `all` or `versions`. Setting it to `versions` enables extremely fast replication (next second!) of data modifications.

- `dynamicFetchAddr`: Defines local cluster pod service host:port. Used by EdgeFS gateways (S3, S3X, NFS, SMB or iSCSI) to distribute missing chunk fetch requests. It needs to be set to other than 0.0.0.0 on access endpoints so to enable dynamic fetch feature. Default is 0.0.0.0:49678.
- `localAddr`: Defines local cluster pod service address and port. Used by ISGW link endpoint. Kubernetes ExternalIP will be automatically constructed if address is provided. Default is 0.0.0.0:14000.
- `useEncryptedTunnel`: If true, enables usage of encrypted SSH tunnel. By default, transferred metadata and data chunks are just compressed but unencrypted. Enable encryption if ISGW links crossing boundaries of single data center or multiple clouds.
- `chunkCacheSize`: Limit amount of memory allocated for dynamic chunk cache. By default ISGW pod uses up to 75% of available memory as chunk caching area. This option can influence this allocation strategy.
- `config`: Advanced ISGW services configurations - one kubernetes service per many EdgeFS services. `Server` is EdgeFS ISGW server service name, `Clients` array of ISGW client services names. When `config` option is defined the `remoteURL` is optional.
- `annotations`: Key value pair list of annotations to add.
- `placement`: The ISGW pods can be given standard Kubernetes placement restrictions with `nodeAffinity`, `tolerations`, `podAffinity`, and `podAntiAffinity` similar to placement defined for daemons configured by the cluster CRD.
- `resourceProfile`: ISGW pod resource utilization profile (Memory and CPU). Can be `embedded` or `performance` (default). In case of `performance` an ISGW pod trying to increase amount of internal I/O resources that results in higher performance at the cost of additional memory allocation and more CPU load. In `embedded` profile case, ISGW pod gives preference to preserving memory over I/O and limiting chunk cache (see `chunkCacheSize` option). The `performance` profile is the default unless cluster wide `embedded` option is defined.
- `resources`: Set resource requests/limits for the ISGW pods, see Resource Requirements/Limits.

Setting up two-sites bi-directional syncing EdgeFS namespace and tenants

This example will demonstrate creation of simple syncing namespace between two sites.

For more detailed instructions please refer to EdgeFS Wiki.

Below is an exampmle procedure to get things initialized and configured.

Before new local namespace (or local site) can be used, it has to be initialized with FlexHash and special purpose root object.

FlexHash consists of dynamically discovered configuration and checkpoint of accepted distribution table. FlexHash is responsible for I/O direction and plays important role in dynamic load balancing logic. It defines so-called Negotiating Groups (typically across zoned 8-24 disks) and final table distribution across all the participating components, e.g. data nodes, service gateways and tools.

Root object holds system information and table of namespaces registered to a local site. Root object is always local and never shared between the sites.

For each cluster, to initialize system and prepare logical definitions, login to the toolbox as shown in this example:

```
1. kubectl get po --all-namespaces | grep edgefs-mgr
   kubectl exec -it -n rook-edgefs rook-edgefs-mgr-6cb9598469-czr7p -- env COLUMNS=$COLUMNS LINES=$LINES
2. TERM=linux toolbox
```

Assumption at this point is that nodes are all configured and can be seen via the following command:

```
1. efscli system status
```

1. Initialize EdgeFS cluster:

Verify that HW (or better say emulated in this case) configuration look normal and accept it

```
1. efscli system init
```

At this point new dynamically discovered configuration checkpoint will be created at `$NEDGE_HOME/var/run/flexhash-checkpoint.json`. This will also create system “root” object, holding Site’s Namespace. Namespace may consist of more then single region.

1. Create new local namespace (or we also call it “Region” or “Segment”):

```
1. efscli cluster create Hawaii
```

1. Create logical tenants of cluster namespace “Hawaii”, also buckets if needed:

```
1. efscli tenant create Hawaii/Cola
2. efscli bucket create Hawaii/Cola/bk1
3. efscli tenant create Hawaii/Pepsi
4. efscli bucket create Hawaii/Pepsi/bk1
```

Now that Hawaii serving namespace cluster is setup, repeat steps on the secondary

site:

```
efscli cluster create Hawaii
efscli tenant create Hawaii/Cola -r 2
efscli tenant create Hawaii/Pepsi -r 2
```

This steps also includes a possibility to override destination defaults on per-tenant (Cola and Pepsi) basis. In the example above we setting replication count to 2 on the secondary site.

1. Create ISGW link services on both EdgeFS sites:

Order of creation of ISGW links doesn’t matter. Link status change will be noticed automatically and syncing can be delayed for up to 7 days by default.

On the primary site:

```
efscli service create isgw hawaii
efscli service serve hawaii Hawaii/Cola/bk1
efscli service serve hawaii Hawaii/Pepsi/bk1
```

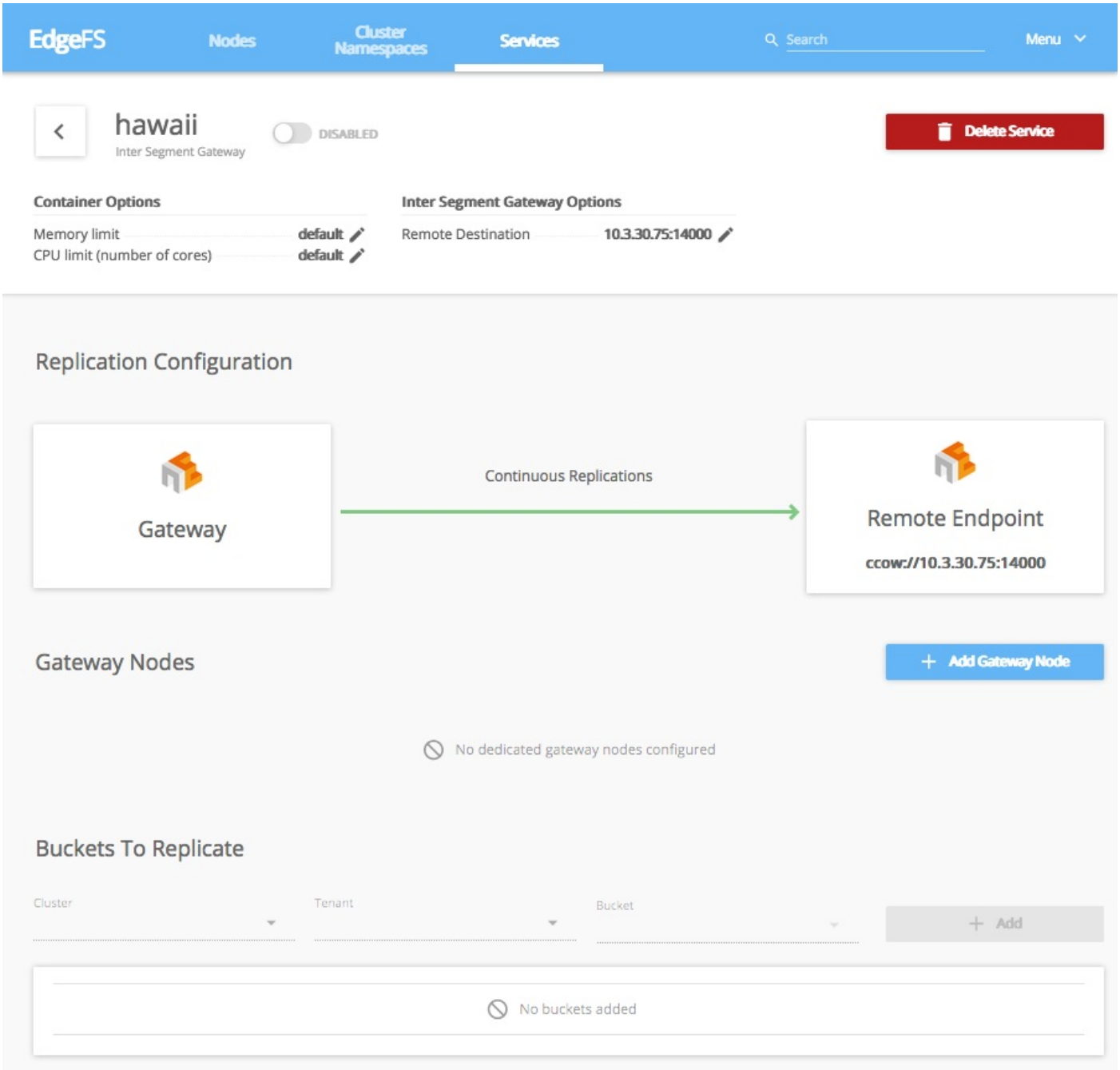
Because we creating bi-directional link, on the secondary site we would need to create exactly same configuration except X-ISGW-Remote (remoteURL in CRD) where it should point back to primary. Synchronization is enabled on per-bucket level. Buckets which do not yet exist will be automatically created with tenant parameter overrides taken into an account.

1. Create ISGW Link CRDs on both EdgeFS sites:

```
apiVersion: edgefs.rook.io/v1
kind: ISGW
metadata:
  name: hawaii
  namespace: rook-edgefs
spec:
  direction: send+receive
  remoteURL: ccow://10.3.32.240:14000
  localAddr: 10.3.30.75:14000
```

```
apiVersion: edgefs.rook.io/v1
kind: ISGW
metadata:
  name: hawaii
  namespace: rook-edgefs
spec:
  direction: send+receive
  remoteURL: ccow://10.3.30.75:14000
  localAddr: 10.3.32.240:14000
```

It also maybe a bit easier to create services and CRDs using EdgeFS UI Dashboard. ISGW Link page may look like this:



EdgeFS UI Dashboard automates the process of CRD composing but also has a capability to execute and monitor running Kubernetes Rook CRDs.

At this point two ISGW link services should be operational. Verify its operation with `show` command:

```
1. efscli service show hawaii -s
```

EdgeFS Scale-Out NFS CRD

Rook allows creation and customization of EdgeFS NFS filesystems through the custom resource definitions (CRDs). The following settings are available for customization of EdgeFS NFS services.

Sample

```
apiVersion: edgefs.rook.io/v1
kind: NFS
metadata:
  name: nfs01
  namespace: rook-edgefs
spec:
  instances: 3
  #relaxedDirUpdates: true
  #chunkCacheSize: 1Gi
  # A key/value list of annotations
  annotations:
  #  key: value
  placement:
  #  nodeAffinity:
  #    requiredDuringSchedulingIgnoredDuringExecution:
  #      nodeSelectorTerms:
  #      - matchExpressions:
  #        - key: role
  #          operator: In
  #          values:
  #          - nfs-node
  #  tolerations:
  #  - key: nfs-node
  #    operator: Exists
  #  podAffinity:
  #  podAntiAffinity:
  #resourceProfile: embedded
  resources:
  #  limits:
  #    cpu: "500m"
  #    memory: "1024Mi"
  #  requests:
  #    cpu: "500m"
  #    memory: "1024Mi"
```

Metadata

- `name`: The name of the NFS system to create, which must match existing EdgeFS service.

- `namespace`: The namespace of the Rook cluster where the NFS service is created.
- `instances`: The number of active NFS service instances. EdgeFS NFS service is Multi-Head capable, such so that multiple PODs can mount same tenant's buckets via different endpoints. EdgeFS CSI provisioner orchestrates distribution and load balancing across NFS service instances in round-robin or random policy ways.
- `relaxedDirUpdates`: If set to `true` then it will significantly improve performance of directory operations by defering updates, guaranteeing eventual directory consistency. This option is recommended when a bucket exported via single NFS instance and it is not a destination for ISGW Link synchronization.
- `chunkCacheSize`: Limit amount of memory allocated for dynamic chunk cache. By default NFS pod uses up to 75% of available memory as chunk caching area. This option can influence this allocation strategy.
- `annotations`: Key value pair list of annotations to add.
- `placement`: The NFS PODs can be given standard Kubernetes placement restrictions with `nodeAffinity`, `tolerations`, `podAffinity`, and `podAntiAffinity` similar to placement defined for daemons configured by the cluster CRD.
- `resourceProfile`: NFS pod resource utilization profile (Memory and CPU). Can be `embedded` or `performance` (default). In case of `performance` an NFS pod trying to increase amount of internal I/O resources that results in higher performance at the cost of additional memory allocation and more CPU load. In `embedded` profile case, NFS pod gives preference to preserving memory over I/O and limiting chunk cache (see `chunkCacheSize` option). The `performance` profile is the default unless cluster wide `embedded` option is defined.
- `resources`: Set resource requests/limits for the NFS Pod(s), see Resource Requirements/Limits.

Setting up EdgeFS namespace and tenant

For more detailed instructions please refer to EdgeFS Wiki.

Below is an exampmle procedure to get things initialized and configured.

Before new local namespace (or local site) can be used, it has to be initialized with FlexHash and special purpose root object.

FlexHash consists of dynamically discovered configuration and checkpoint of accepted distribution table. FlexHash is responsible for I/O direction and plays important role in dynamic load balancing logic. It defines so-called Negotiating Groups (typically across zoned 8-24 disks) and final table distribution across all the participating components, e.g. data nodes, service gateways and tools.

Root object holds system information and table of namespaces registered to a local site. Root object is always local and never shared between the sites.

To initialize system and prepare logical definitions, login to the toolbox as shown in this example:

```
kubectl get po --all-namespaces | grep edgefs-mgr
kubectl exec -it -n rook-edgefs rook-edgefs-mgr-6cb9598469-czr7p -- env COLUMNS=$COLUMNS LINES=$LINES
TERM=linux toolbox
```

Assumption at this point is that nodes are all configured and can be seen via the following command:

```
efscli system status
```

1. Initialize EdgeFS cluster:

Verify that HW (or better say emulated in this case) configuration look normal and accept it:

```
efscli system init
```

At this point new dynamically discovered configuration checkpoint will be created at $NEDGE_HOME/var/run/flexhash-checkpoint.json This will also create system “root” object, holding Site’s Namespace. Namespace may consist of more then single region.

1. Create new local namespace (or we also call it “Region” or “Segment”):

```
efscli cluster create Hawaii
```

1. Create logical tenants of cluster namespace “Hawaii”, also buckets if needed:

```
efscli tenant create Hawaii/Cola
efscli bucket create Hawaii/Cola/bk1
efscli tenant create Hawaii/Pepsi
efscli bucket create Hawaii/Pepsi/bk1
```

Now cluster is setup, services can be now created and attached to CSI provisioner.

1. Create NFS service objects for tenants:

```
efscli service create nfs nfs-cola
efscli service serve nfs-cola Hawaii/Cola/bk1
efscli service create nfs nfs-pepsi
efscli service serve nfs-pepsi Hawaii/Pepsi/bk1
```

1. Create your EdgeFS NFS objects:

```
apiVersion: edgefs.rook.io/v1
kind: NFS
metadata:
  name: nfs-cola
  namespace: rook-edgefs
```

```
spec:
  instances: 1
```

```
apiVersion: edgefs.rook.io/v1
kind: NFS
metadata:
  name: nfs-pepsi
  namespace: rook-edgefs
spec:
  instances: 1
```

At this point two NFS services should be available. Verify that showmount command can see service (substitute CLUSTERIP with corresponding entry from `kubectl get svc` command):

```
kubectl get svc --all-namespaces
showmount -e CLUSTERIP
```

EdgeFS Scale-Out SMB CRD

Rook allows creation and customization of EdgeFS SMB/CIFS filesystems through the custom resource definitions (CRDs). The following settings are available for customization of EdgeFS SMB services.

Sample

```
apiVersion: edgefs.rook.io/v1
kind: SMB
metadata:
  name: smb01
  namespace: rook-edgefs
spec:
  instances: 3
  #ads:
  #  domainName: "corp.example.com"
  #  dcName: "localdc"
  #  serverName: "edgefs-smb"
  #  userSecret: "corp.example.com"
  #  nameservers: "10.200.1.19"
  #relaxedDirUpdates: true
  #chunkCacheSize: 1Gi
  # A key/value list of annotations
  annotations:
  #  key: value
  placement:
  #  nodeAffinity:
  #    requiredDuringSchedulingIgnoredDuringExecution:
  #      nodeSelectorTerms:
  #      - matchExpressions:
  #        - key: role
  #          operator: In
  #          values:
  #          - smb-node
  #  tolerations:
  #  - key: smb-node
  #    operator: Exists
  #  podAffinity:
  #  podAntiAffinity:
  #resourceProfile: embedded
  resources:
  #  limits:
  #    cpu: "500m"
  #    memory: "1024Mi"
  #  requests:
  #    cpu: "500m"
  #    memory: "1024Mi"
```

Metadata

- `name`: The name of the SMB system to create, which must match existing EdgeFS service.
- `namespace`: The namespace of the Rook cluster where the SMB service is created.
- `instances`: The number of active SMB service instances. EdgeFS SMB service is Multi-Head capable, such so that multiple PODs can mount same tenant's buckets via different endpoints.
- `relaxedDirUpdates`: If set to `true` then it will significantly improve performance of directory operations by defering updates, guaranteeing eventual directory consistency. This option is recommended when a bucket exported via single SMB instance and it is not a destination for ISGW Link synchronization.
- `chunkCacheSize`: Limit amount of memory allocated for dynamic chunk cache. By default SMB pod uses up to 75% of available memory as chunk caching area. This option can influence this allocation strategy.
- `annotations`: Key value pair list of annotations to add.
- `placement`: The SMB PODs can be given standard Kubernetes placement restrictions with `nodeAffinity`, `tolerations`, `podAffinity`, and `podAntiAffinity` similar to placement defined for daemons configured by the cluster CRD.
- `resourceProfile`: SMB pod resource utilization profile (Memory and CPU). Can be `embedded` or `performance` (default). In case of `performance` an SMB pod trying to increase amount of internal I/O resources that results in higher performance at the cost of additional memory allocation and more CPU load. In `embedded` profile case, SMB pod gives preference to preserving memory over I/O and limiting chunk cache (see `chunkCacheSize` option). The `performance` profile is the default unless cluster wide `embedded` option is defined.
- `resources`: Set resource requests/limits for the SMB Pod(s), see Resource Requirements/Limits.
- `ads`: Set Active Directory service join parameters. If not defined, SMB gateway will start in WORKGROUP mode.

Joining Windows Active Directory service

Before you begin, please make sure that external IP can be properly provisioned to passthrough ports 445 (smb) and 139 (netbios), pointing to SMB service. If for some reason external IP is difficult or impossible to provision, you can use NodePort and setup redirect rules on the node where SMB gateway will be running.

Define "ads" metadata section with the following parameters and reference a secret object:

- `domainName`: AD Domain Name in form of DNS record, like `corp.example.com`.
- `dcName`: Preferred Domain Controller Name. Could be short name like `localdc`.
- `serverName`: NetBIOS Name of our SMB Gateway. This name will be used to during SMB share mapping, e.g. `\\edgefs-smb\bk1`.
- `userSecret`: The name of secret holding username and password keys. Secret object

has to be pre-created in the same namespace.

- `nameservers`: The comma separated list of DNS name server IPs, e.g. `10.3.40.16,10.3.40.17`

Secret object can look like this:

```
apiVersion: v1
kind: Secret
metadata:
  name: corp.example.com
  namespace: rook-edgefs
type: Opaque
stringData:
  username: "Administrator"
  password: "Password!"
```

Setting up EdgeFS namespace and tenant

For more detailed instructions please refer to EdgeFS Wiki.

Below is an exampmle procedure to get things initialized and configured.

Before new local namespace (or local site) can be used, it has to be initialized with FlexHash and special purpose root object.

FlexHash consists of dynamically discovered configuration and checkpoint of accepted distribution table. FlexHash is responsible for I/O direction and plays important role in dynamic load balancing logic. It defines so-called Negotiating Groups (typically across zoned 8-24 disks) and final table distribution across all the participating components, e.g. data nodes, service gateways and tools.

Root object holds system information and table of namespaces registered to a local site. Root object is always local and never shared between the sites.

To initialize system and prepare logical definitions, login to the toolbox as shown in this example:

```
kubectl get po --all-namespaces | grep edgefs-mgr
kubectl exec -it -n rook-edgefs rook-edgefs-mgr-6cb9598469-czr7p -- env COLUMNS=$COLUMNS LINES=$LINES
TERM=linux toolbox
```

Assumption at this point is that nodes are all configured and can be seen via the following command:

```
efscli system status
```

1. Initialize EdgeFS cluster:

Verify that HW (or better say emulated in this case) configuration look normal and accept it:

```
1. efscli system init
```

At this point new dynamically discovered configuration checkpoint will be created at $NEDGE_HOME/var/run/flexhash-checkpoint.json This will also create system “root” object, holding Site’s Namespace. Namespace may consist of more then single region.

1. Create new local namespace (or we also call it “Region” or “Segment”):

```
1. efscli cluster create Hawaii
```

1. Create logical tenants of cluster namespace “Hawaii”, also buckets if needed:

```
1. efscli tenant create Hawaii/Cola
2. efscli bucket create Hawaii/Cola/bk1
3. efscli tenant create Hawaii/Pepsi
4. efscli bucket create Hawaii/Pepsi/bk1
```

Now cluster is setup, services can be now created and attached to CSI provisioner.

1. Create SMB service objects for tenants:

```
1. efscli service create smb smb-cola
2. efscli service serve smb-cola Hawaii/Cola/bk1
   efscli service config smb-cola X-SMB-OPTS-Cola-bk1 "force user = root;public = yes;directory mask = 777;create
3. mask = 666"
4. efscli service create smb smb-pepsi
5. efscli service serve smb-pepsi Hawaii/Pepsi/bk1
   efscli service config smb-pepsi X-SMB-OPTS-Pepsi-bk1 "force user = root;public = yes;directory mask =
6. 777;create mask = 666"
```

Also, notice that we setting password-less configuration for bk1 share.

1. Create your EdgeFS SMB objects:

```
1. apiVersion: edgefs.rook.io/v1
2. kind: SMB
3. metadata:
4.   name: smb-cola
5.   namespace: rook-edgefs
6. spec:
7.   instances: 1
```

```
1. apiVersion: edgefs.rook.io/v1
2. kind: SMB
3. metadata:
4.   name: smb-pepsi
```

```
  namespace: rook-edgefs
spec:
  instances: 1
```

At this point two SMB services should be available. Install cifs-utils package and verify that you can mount it (substitute CLUSTERIP with corresponding entry from `kubectl get svc` command):

```
kubectl get -o rook-edgefs svc | grep smb-pepsi
mount -t cifs -o vers=3.0,sec=none //CLUSTERIP/bk1 /tmp/bk1
```

Edge-X S3 CRD

The API provides access to advanced EdgeFS Object interfaces, such as access to Key-Value store, S3 Object Append mode, S3 Object RW mode and S3 Object Stream Session (POSIX compatible) mode. A Stream Session encompasses a series of edits to one object made by one source that are saved as one or more versions during a specific finite time duration. A Stream Session must be isolated while it is open. That is, users working through this session will not see updates to this object from other sessions.Stream Session allows high-performance POSIX-style access to an object and thus it is beneficial for client applications to use HTTP/1.1 Persistent Connection extensions, to minimize latency between updates or reads.

For more detailes on API please refer to Edge-S3 API.

Rook allows creation and customization of Edge-X S3 services through the custom resource definitions (CRDs). The following settings are available for customization of Edge-S3 services.

Sample

```
apiVersion: edgefs.rook.io/v1
kind: S3X
metadata:
  name: s3x01
  namespace: rook-edgefs
spec:
  instances: 3
  #chunkCacheSize: 1Gi
  # A key/value list of annotations
  annotations:
  #  key: value
  placement:
  #  nodeAffinity:
  #    requiredDuringSchedulingIgnoredDuringExecution:
  #      nodeSelectorTerms:
  #      - matchExpressions:
  #        - key: role
  #          operator: In
  #          values:
  #          - s3x-node
  #  tolerations:
  #  - key: s3x-node
  #    operator: Exists
  #  podAffinity:
  #  podAntiAffinity:
  #resourceProfile: embedded
  resources:
  #  limits:
```

```
    #    cpu: "500m"
    #    memory: "1024Mi"
    #  requests:
    #    cpu: "500m"
    #    memory: "1024Mi"
```

Metadata

- `name`: The name of the Edge-X S3 system to create, which must match existing EdgeFS service.
- `namespace`: The namespace of the Rook cluster where the Edge-X S3 service is created.
- `sslCertificateRef`: If the certificate is not specified, SSL will use default crt and key files. If specified, this is the name of the Kubernetes secret that contains the SSL certificate to be used for secure connections. Please see secret YAML file example on how to setup Kuberenetes secret. Notice that base64 encoding is required.
- `port`: The port on which the Edge-X S3 pods and the Edge-X S3 service will be listening (not encrypted). Default port is 3000.
- `securePort`: The secure port on which Edge-X S3 pods will be listening. If not defined then default SSL certificates will be used. Default port is 3001.
- `instances`: The number of active Edge-X S3 service instances. For load balancing we recommend to use nginx and the like solutions.
- `chunkCacheSize`: Limit amount of memory allocated for dynamic chunk cache. By default S3X pod uses up to 75% of available memory as chunk caching area. This option can influence this allocation strategy.
- `annotations`: Key value pair list of annotations to add.
- `placement`: The Edge-X S3 PODs can be given standard Kubernetes placement restrictions with `nodeAffinity`, `tolerations`, `podAffinity`, and `podAntiAffinity` similar to placement defined for daemons configured by the cluster CRD.
- `resourceProfile`: S3X pod resource utilization profile (Memory and CPU). Can be `embedded` or `performance` (default). In case of `performance` an S3X pod trying to increase amount of internal I/O resources that results in higher performance at the cost of additional memory allocation and more CPU load. In `embedded` profile case, S3X pod gives preference to preserving memory over I/O and limiting chunk cache (see `chunkCacheSize` option). The `performance` profile is the default unless cluster wide `embedded` option is defined.
- `resources`: Set resource requests/limits for the Edge-X S3 Pod(s), see Resource Requirements/Limits.

Setting up EdgeFS namespace and tenant

For more detailed instructions please refer to EdgeFS Wiki.

Below is an exampmle procedure to get things initialized and configured.

Before new local namespace (or local site) can be used, it has to be initialized with FlexHash and special purpose root object.

FlexHash consists of dynamically discovered configuration and checkpoint of accepted distribution table. FlexHash is responsible for I/O direction and plays important role in dynamic load balancing logic. It defines so-called Negotiating Groups (typically across zoned 8-24 disks) and final table distribution across all the participating components, e.g. data nodes, service gateways and tools.

Root object holds system information and table of namespaces registered to a local site. Root object is always local and never shared between the sites.

To initialize system and prepare logical definitions, login to the toolbox as shown in this example:

```
1. kubectl get po --all-namespaces | grep edgefs-mgr
   kubectl exec -it -n rook-edgefs rook-edgefs-mgr-6cb9598469-czr7p -- env COLUMNS=$COLUMNS LINES=$LINES
2. TERM=linux toolbox
```

Assumption at this point is that nodes are all configured and can be seen via the following command:

```
1. efscli system status
```

1. Initialize EdgeFS cluster:

Verify that HW (or better say emulated in this case) configuration look normal and accept it

```
1. efscli system init
```

At this point new dynamically discovered configuration checkpoint will be created at $NEDGE_HOME/var/run/flexhash-checkpoint.json This will also create system “root” object, holding Site’s Namespace. Namespace may consist of more then single region.

1. Create new local namespace (or we also call it “Region” or “Segment”):

```
1. efscli cluster create Hawaii
```

1. Create logical tenants of cluster namespace “Hawaii”, also buckets if needed:

```
1. efscli tenant create Hawaii/Cola
2. efscli bucket create Hawaii/Cola/bk1
3. efscli tenant create Hawaii/Pepsi
4. efscli bucket create Hawaii/Pepsi/bk1
```

Now cluster is setup, services can be now created.

1. Create Edge-X S3 services objects for tenants:

```
efscli service create s3x s3x-cola
efscli service serve s3x-cola Hawaii/Cola
efscli service create s3x s3x-pepsi
efscli service serve s3x-pepsi Hawaii/Pepsi
```

1. Create EdgeFS S3X CRDs:

```
apiVersion: edgefs.rook.io/v1
kind: S3X
metadata:
  name: s3x-cola
  namespace: rook-edgefs
spec:
  instances: 1
```

```
apiVersion: edgefs.rook.io/v1
kind: S3X
metadata:
  name: s3x-pepsi
  namespace: rook-edgefs
spec:
  instances: 1
```

At this point two Edge-X S3 services should be available and listening on default ports.

EdgeFS AWS S3 CRD

Rook allows creation and customization of AWS S3 compatible services through the custom resource definitions (CRDs). The following settings are available for customization of S3 services.

Sample

```
apiVersion: edgefs.rook.io/v1
kind: S3
metadata:
  name: s301
  namespace: rook-edgefs
spec:
  instances: 3
  #s3type: s3s
  #chunkCacheSize: 1Gi
  # A key/value list of annotations
  annotations:
  #  key: value
  placement:
  #  nodeAffinity:
  #    requiredDuringSchedulingIgnoredDuringExecution:
  #      nodeSelectorTerms:
  #      - matchExpressions:
  #        - key: role
  #          operator: In
  #          values:
  #          - s3-node
  #  tolerations:
  #  - key: s3-node
  #    operator: Exists
  #  podAffinity:
  #  podAntiAffinity:
  #resourceProfile: embedded
  resources:
  #  limits:
  #    cpu: "500m"
  #    memory: "1024Mi"
  #  requests:
  #    cpu: "500m"
  #    memory: "1024Mi"
```

Metadata

- `name`: The name of the S3 system to create, which must match existing EdgeFS service.

- `namespace`: The namespace of the Rook cluster where the S3 service is created.
- `s3type`: The type of S3 service to be created. It can be one of the following: `s3` (default, path style) or `s3s` (buckets as DNS style)
- `sslCertificateRef`: If the certificate is not specified, SSL will use default crt and key files. If specified, this is the name of the Kubernetes secret that contains the SSL certificate to be used for secure connections. Please see secret YAML file example on how to setup Kuberenetes secret. Notice that base64 encoding is required.
- `port`: The port on which the S3 pods and the S3 service will be listening (not encrypted). Default port is 9982 for `s3` and 9983 for `s3s`.
- `securePort`: The secure port on which S3 pods will be listening. If not defined then default SSL certificates will be used. Default port is 8443 for `s3` and 8444 for `s3s`.
- `chunkCacheSize`: Limit amount of memory allocated for dynamic chunk cache. By default S3 pod uses up to 75% of available memory as chunk caching area. This option can influence this allocation strategy.
- `instances`: The number of active S3 service instances. For load balancing we recommend to use nginx and the like solutions.
- `annotations`: Key value pair list of annotations to add.
- `placement`: The S3 pods can be given standard Kubernetes placement restrictions with `nodeAffinity`, `tolerations`, `podAffinity`, and `podAntiAffinity` similar to placement defined for daemons configured by the cluster CRD.
- `resourceProfile`: S3 pod resource utilization profile (Memory and CPU). Can be `embedded` or `performance` (default). In case of `performance` an S3 pod trying to increase amount of internal I/O resources that results in higher performance at the cost of additional memory allocation and more CPU load. In `embedded` profile case, S3 pod gives preference to preserving memory over I/O and limiting chunk cache (see `chunkCacheSize` option). The `performance` profile is the default unless cluster wide `embedded` option is defined.
- `resources`: Set resource requests/limits for the S3 pods, see Resource Requirements/Limits.

Setting up EdgeFS namespace and tenant

For more detailed instructions please refer to EdgeFS Wiki.

Below is an exampmle procedure to get things initialized and configured.

Before new local namespace (or local site) can be used, it has to be initialized with FlexHash and special purpose root object.

FlexHash consists of dynamically discovered configuration and checkpoint of accepted distribution table. FlexHash is responsible for I/O direction and plays important role in dynamic load balancing logic. It defines so-called Negotiating Groups (typically across zoned 8-24 disks) and final table distribution across all the participating components, e.g. data nodes, service gateways and tools.

Root object holds system information and table of namespaces registered to a local site. Root object is always local and never shared between the sites.

To initialize system and prepare logical definitions, login to the toolbox as shown in this example:

```
kubectl get po --all-namespaces | grep edgefs-mgr
kubectl exec -it -n rook-edgefs rook-edgefs-mgr-6cb9597469-czr7p -- env COLUMNS=$COLUMNS LINES=$LINES
TERM=linux toolbox
```

Assumption at this point is that nodes are all configured and can be seen via the following command:

```
efscli system status
```

1. Initialize EdgeFS cluster:

Verify that HW (or better say emulated in this case) configuration look normal and accept it

```
efscli system init
```

At this point new dynamically discovered configuration checkpoint will be created at $NEDGE_HOME/var/run/flexhash-checkpoint.json This will also create system “root” object, holding Site’s Namespace. Namespace may consist of more then single region.

1. Create new local namespace (or we also call it “Region” or “Segment”):

```
efscli cluster create Hawaii
```

1. Create logical tenants of cluster namespace “Hawaii”, also buckets if needed:

```
efscli tenant create Hawaii/Cola
efscli bucket create Hawaii/Cola/bk1
efscli tenant create Hawaii/Pepsi
efscli bucket create Hawaii/Pepsi/bk1
```

Now cluster is setup, services can be now created.

1. Create S3 services objects for tenants:

```
efscli service create s3 s3-cola
efscli service serve s3-cola Hawaii/Cola
efscli service create s3 s3-pepsi
efscli service serve s3-pepsi Hawaii/Pepsi
```

In case of s3type set to `s3`, do not forget to configure default domain name:

```
efscli service config s3-cola X-Domain cola.com
efscli service config s3-pepsi X-Domain pepsi.com
```

1. Create EdgeFS S3 objects:

```
apiVersion: edgefs.rook.io/v1
kind: S3
metadata:
  name: s3-cola
  namespace: rook-edgefs
spec:
  instances: 1
```

```
apiVersion: edgefs.rook.io/v1
kind: S3
metadata:
  name: s3-pepsi
  namespace: rook-edgefs
spec:
  instances: 1
```

At this point two S3 services should be available and listening on default ports.

EdgeFS OpenStack/SWIFT CRD

Rook allows creation and customization of OpenStack/SWIFT compatible services through the custom resource definitions (CRDs). The following settings are available for customization of SWIFT services.

Sample

```
apiVersion: edgefs.rook.io/v1
kind: SWIFT
metadata:
  name: swift01
  namespace: rook-edgefs
spec:
  instances: 3
  #chunkCacheSize: 1Gi
  # A key/value list of annotations
  annotations:
  #  key: value
  placement:
  #  nodeAffinity:
  #    requiredDuringSchedulingIgnoredDuringExecution:
  #      nodeSelectorTerms:
  #      - matchExpressions:
  #        - key: role
  #          operator: In
  #          values:
  #          - swift-node
  #  tolerations:
  #  - key: swift-node
  #    operator: Exists
  #  podAffinity:
  #  podAntiAffinity:
  #resourceProfile: embedded
  resources:
  #  limits:
  #    cpu: "500m"
  #    memory: "1024Mi"
  #  requests:
  #    cpu: "500m"
  #    memory: "1024Mi"
```

Metadata

- `name`: The name of the SWIFT service to create, which must match existing EdgeFS service.
- `namespace`: The namespace of the Rook cluster where the SWIFT service is created.

- `sslCertificateRef`: If the certificate is not specified, SSL will use default crt and key files. If specified, this is the name of the Kubernetes secret that contains the SSL certificate to be used for secure connections. Please see secret YAML file example on how to setup Kuberenetes secret. Notice that base64 encoding is required.
- `port`: The port on which the SWIFT pods and the SWIFT service will be listening (not encrypted). Default port is 9981.
- `securePort`: The secure port on which SWIFT pods will be listening. If not defined then default SSL certificates will be used. Default port is 443.
- `instances`: The number of active SWIFT service instances. For load balancing we recommend to use nginx and the like solutions.
- `chunkCacheSize`: Limit amount of memory allocated for dynamic chunk cache. By default SWIFT pod uses up to 75% of available memory as chunk caching area. This option can influence this allocation strategy.
- `annotations`: Key value pair list of annotations to add.
- `placement`: The SWIFT pods can be given standard Kubernetes placement restrictions with `nodeAffinity`, `tolerations`, `podAffinity`, and `podAntiAffinity` similar to placement defined for daemons configured by the cluster CRD.
- `resourceProfile`: SWIFT pod resource utilization profile (Memory and CPU). Can be `embedded` or `performance` (default). In case of `performance` an SWIFT pod trying to increase amount of internal I/O resources that results in higher performance at the cost of additional memory allocation and more CPU load. In `embedded` profile case, SWIFT pod gives preference to preserving memory over I/O and limiting chunk cache (see `chunkCacheSize` option). The `performance` profile is the default unless cluster wide `embedded` option is defined.
- `resources`: Set resource requests/limits for the SWIFT pods, see Resource Requirements/Limits.

Setting up EdgeFS namespace and tenant

For more detailed instructions please refer to EdgeFS Wiki.

Below is an exampmle procedure to get things initialized and configured.

Before new local namespace (or local site) can be used, it has to be initialized with FlexHash and special purpose root object.

FlexHash consists of dynamically discovered configuration and checkpoint of accepted distribution table. FlexHash is responsible for I/O direction and plays important role in dynamic load balancing logic. It defines so-called Negotiating Groups (typically across zoned 8-24 disks) and final table distribution across all the participating components, e.g. data nodes, service gateways and tools.

Root object holds system information and table of namespaces registered to a local site. Root object is always local and never shared between the sites.

To initialize system and prepare logical definitions, login to the toolbox as shown in

this example:

```
kubectl get po --all-namespaces | grep edgefs-mgr
kubectl exec -it -n rook-edgefs rook-edgefs-mgr-6cb9598469-czr7p -- env COLUMNS=$COLUMNS LINES=$LINES
TERM=linux toolbox
```

Assumption at this point is that nodes are all configured and can be seen via the following command:

```
efscli system status
```

1. Initialize EdgeFS cluster:

Verify that HW (or better say emulated in this case) configuration look normal and accept it

```
efscli system init
```

At this point new dynamically discovered configuration checkpoint will be created at $NEDGE_HOME/var/run/flexhash-checkpoint.json This will also create system “root” object, holding Site’s Namespace. Namespace may consist of more then single region.

1. Create new local namespace (or we also call it “Region” or “Segment”):

```
efscli cluster create Hawaii
```

1. Create logical tenants of cluster namespace “Hawaii”, also buckets if needed:

```
efscli tenant create Hawaii/Cola
efscli bucket create Hawaii/Cola/bk1
efscli tenant create Hawaii/Pepsi
efscli bucket create Hawaii/Pepsi/bk1
```

Now cluster is setup, services can be now created.

1. Create SWIFT services objects for tenants:

```
efscli service create swift swift-cola
efscli service serve swift-cola Hawaii
efscli service create swift swiftPepsi
efscli service serve swift-pepsi Hawaii
```

1. Create EdgeFS SWIFT objects:

```
apiVersion: edgefs.rook.io/v1
kind: SWIFT
metadata:
```

```
    name: swiftCola
    namespace: rook-edgefs
spec:
    instances: 1
```

```
apiVersion: edgefs.rook.io/v1
kind: SWIFT
metadata:
    name: swiftPepsi
    namespace: rook-edgefs
spec:
    instances: 1
```

At this point two SWIFT services should be available and listening on default ports.

EdgeFS iSCSI Target CRD

Rook allows creation and customization of High Performance iSCSI Target compatible services through the custom resource definitions (CRDs). The following settings are available for customization of iSCSI Target services.

Sample

```
apiVersion: edgefs.rook.io/v1
kind: ISCSI
metadata:
  name: iscsi01
  namespace: rook-edgefs
spec:
  #chunkCacheSize: 1Gi
  placement:
  #  nodeAffinity:
  #    requiredDuringSchedulingIgnoredDuringExecution:
  #      nodeSelectorTerms:
  #      - matchExpressions:
  #        - key: role
  #          operator: In
  #          values:
  #          - iscsi-node
  #  tolerations:
  #  - key: iscsi-node
  #    operator: Exists
  #  podAffinity:
  #  podAntiAffinity:
  #resourceProfile: embedded
  resources:
  #  limits:
  #    cpu: "500m"
  #    memory: "1024Mi"
  #  requests:
  #    cpu: "500m"
  #    memory: "1024Mi"
  # A key/value list of annotations
  annotations:
  #  key: value
```

Metadata

- `name`: The name of the iSCSI target service to create, which must match existing EdgeFS service.
- `namespace`: The namespace of the Rook cluster where the iSCSI Target service is

created.

- `targetName`: The name for iSCSI target name. Default is iqn.2018-11.edgefs.io.
- `targetParams`: If specified, then some of iSCSI target protocol parameters can be overridden.
  - `MaxRecvDataSegmentLength`: Value in range value range 512..16777215. Default is 524288.
  - `DefaultTime2Retain`: Value in range 0..3600. Default is 60.
  - `DefaultTime2Wait`: Value in range 0..3600. Default is 30.
  - `FirstBurstLength`: Value in range 512..16777215. Default is 524288.
  - `MaxBurstLength`: Value in range 512..16777215. Default is 1048576.
  - `MaxQueueCmd`: Value in range 1..128. Default is 64.
- `chunkCacheSize`: Limit amount of memory allocated for dynamic chunk cache. By default iSCSI pod uses up to 75% of available memory as chunk caching area. This option can influence this allocation strategy.
- `annotations`: Key value pair list of annotations to add.
- `placement`: The iSCSI pods can be given standard Kubernetes placement restrictions with `nodeAffinity`, `tolerations`, `podAffinity`, and `podAntiAffinity` similar to placement defined for daemons configured by the cluster CRD.
- `resourceProfile`: iSCSI pod resource utilization profile (Memory and CPU). Can be `embedded` or `performance` (default). In case of `performance` an iSCSI pod trying to increase amount of internal I/O resources that results in higher performance at the cost of additional memory allocation and more CPU load. In `embedded` profile case, iSCSI pod gives preference to preserving memory over I/O and limiting chunk cache (see `chunkCacheSize` option). The `performance` profile is the default unless cluster wide `embedded` option is defined.
- `resources`: Set resource requests/limits for the iSCSI pods, see Resource Requirements/Limits.

Setting up EdgeFS namespace and tenant

For more detailed instructions please refer to EdgeFS Wiki.

Below is an exampmle procedure to get things initialized and configured.

Before new local namespace (or local site) can be used, it has to be initialized with FlexHash and special purpose root object.

FlexHash consists of dynamically discovered configuration and checkpoint of accepted distribution table. FlexHash is responsible for I/O direction and plays important role in dynamic load balancing logic. It defines so-called Negotiating Groups (typically across zoned 8-24 disks) and final table distribution across all the participating components, e.g. data nodes, service gateways and tools.

Root object holds system information and table of namespaces registered to a local site. Root object is always local and never shared between the sites.

To initialize system and prepare logical definitions, login to the toolbox as shown in

this example:

```
kubectl get po --all-namespaces | grep edgefs-mgr
kubectl exec -it -n rook-edgefs rook-edgefs-mgr-6cb9598469-czr7p -- env COLUMNS=$COLUMNS LINES=$LINES
TERM=linux toolbox
```

Assumption at this point is that nodes are all configured and can be seen via the following command:

```
efscli system status
```

1. Initialize EdgeFS cluster:

Verify that HW (or better say emulated in this case) configuration look normal and accept it

```
efscli system init
```

At this point new dynamically discovered configuration checkpoint will be created at $NEDGE_HOME/var/run/flexhash-checkpoint.json This will also create system “root” object, holding Site’s Namespace. Namespace may consist of more then single region.

1. Create new local namespace (or we also call it “Region” or “Segment”):

```
efscli cluster create Hawaii
```

1. Create logical tenants of cluster namespace “Hawaii”, also buckets if needed:

```
efscli tenant create Hawaii/Cola
efscli bucket create Hawaii/Cola/bk1
efscli tenant create Hawaii/Pepsi
efscli bucket create Hawaii/Pepsi/bk1
```

Now cluster is setup, services can be now created.

1. Create iSCSI Target services objects for tenants:

```
efscli service create iscsi isc-cola
efscli service serve isc-cola Hawaii/Cola/bk1/lun1 X-volsize=10G,ccow-chunkmap-chunk-size=16384
efscli service serve isc-cola Hawaii/Cola/bk1/lun2 X-volsize=20G,ccow-chunkmap-chunk-size=131072
efscli service create iscsi isc-pepsi
efscli service serve isc-pepsi Hawaii/Pepsi/bk1/lun1 X-volsize=20G
```

1. Create ISCSI CRDs:

```
apiVersion: edgefs.rook.io/v1
kind: ISCSI
```

```
3. metadata:
4.   name: iscCola
5.   namespace: rook-edgefs
```

```
1. apiVersion: edgefs.rook.io/v1
2. kind: ISCSI
3. metadata:
4.   name: iscPepsi
5.   namespace: rook-edgefs
```

At this point two iSCSI Target services should be available and listening on default port 3260.

EdgeFS Rook integrated CSI driver, provisioner, attacher and snapshotter

Container Storage Interface (CSI) driver, provisioner, attacher and snapshotter for EdgeFS Scale-Out NFS/ISCSI services

Overview

EdgeFS CSI plugins implement an interface between CSI enabled Container Orchestrator (CO) and EdgeFS local cluster site. It allows dynamic and static provisioning of EdgeFS NFS exports and ISCSI LUNs, and attaching them to application workloads. With EdgeFS NFS/ISCSI implementation, I/O load can be spread-out across multiple PODs, thus eliminating I/O bottlenecks of classing single-node NFS/ISCSI and providing highly available persistent volumes. Current implementation of EdgeFS CSI plugins was tested in Kubernetes environment (requires Kubernetes 1.13+)

Prerequisites

- Ensure your kubernetes cluster version is 1.13+
- Kubernetes cluster must allow privileged pods, this flag must be set for the API server and the kubelet (instructions):

```
1.  --allow-privileged=true
```

- Required the API server and the kubelet feature gates (instructions):

```
1.  --feature-gates=VolumeSnapshotDataSource=true,CSIDriverRegistry=true
```

- Mount propagation must be enabled, the Docker daemon for the cluster must allow shared mounts (instructions)
- Kubernetes CSI drivers require `CSIDriver` and `CSINodeInfo` resource types to be defined on the cluster. Check if they are already defined:

  ```
  1. kubectl get customresourcedefinition.apiextensions.k8s.io/csidrivers.csi.storage.k8s.io
  2. kubectl get customresourcedefinition.apiextensions.k8s.io/csinodeinfos.csi.storage.k8s.io
  ```

  If the cluster doesn't have "csidrivers" and "csinodeinfos" resource types, create them:

  ```
  1. kubectl create -f https://raw.githubusercontent.com/kubernetes/csi-api/release-1.13/pkg/crd/manifests/csidriver.yaml
  ```

```
    kubectl create -f https://raw.githubusercontent.com/kubernetes/csi-api/release-
2. 1.13/pkg/crd/manifests/csinodeinfo.yaml
```

- Depends on preferred CSI driver type, following utilities must be installed on each Kubernetes node (For Debian/Ubuntu based systems):

```
1. # for NFS
2. apt install -y nfs-common rpcbind
3. # for ISCSI
4. apt install -y open-iscsi
```

EdgeFS CSI drivers configuration

For each driver type (NFS/ISCSI) we have already prepared configuration files examples, there are:

- EdgeFS CSI NFS driver config
- EdgeFS CSI ISCSI driver config

Secret file configuration options example:

```
1. # EdgeFS k8s cluster options
2. k8sEdgefsNamespaces: ["rook-edgefs"]          # edgefs cluster namespace
3. k8sEdgefsMgmtPrefix: rook-edgefs-mgr     # edgefs cluster management prefix
4.
5. # EdgeFS csi operations options
6. cluster: cltest           # substitution edgefs cluster name for csi operations
7. tenant: test              # substitution edgefs tenant name for csi operations
8. #serviceFilter: "nfs01"   # comma delimited list of allowed service names for filtering
9.
10. # EdgeFS GRPC security options
11. username: admin          # edgefs k8s cluster grpc service username
12. password: admin          # edgefs k8s cluster grpc service password
```

Options for NFS and ISCSI configuration files

Name	Description	Default value	Required	Type
k8sEdgefsNamespaces	Array of Kubernetes cluster's namespaces for EdgeFS service discovery	rook-edgefs	true	both
k8sEdgefsMgmtPrefix	Rook EdgeFS cluster mgmt service prefix	rook-edgefs-mgr	true	both
username	EdgeFS gRPC API server privileged user	admin	true	both
password	EdgeFS gRPC API server password	admin	true	both
cluster	EdgeFS cluster namespace also known as		false	both

	'region'			
`tenant`	EdgeFS tenant isolated namespace		false	both
`bucket`	EdgeFS tenant bucket to use as a default		false	ISCSI only
`serviceFilter`	Comma delimited list of allowed service names for filtering	`""` means all services allowed	false	both
`serviceBalancerPolicy`	Service selection policy [`minexportspolicy` , `randomservicepolicy`]	`minexportspolicy`	false	both
`chunksize`	Chunk size for actual volume, in bytes	`16384` , should be power of two	false	both
`blocksize`	Block size for actual volume, in bytes	`4096` , should be power of two	false	iSCSI only
`fsType`	New volume's filesystem type	`ext4` , `ext3` , `xfs`	ext4	ISCSI only
`forceVolumeDeletion`	Automatically deletes EdgeFS volume after usage	`false`	false	both

By using `k8sEdgefsNamespaces` and `k8sEdgefsMgmtPrefix` parameters, driver is capable of detecting ClusterIPs and Endpoint IPs to provision and attach volumes.

Apply EdgeFS CSI NFS driver configuration

Check configuration options and create kubernetes secret for Edgefs CSI NFS plugin

```
1. git clone --single-branch --branch release-1.4 https://github.com/rook/rook.git
2. cd rook/cluster/examples/kubernetes/edgefs/csi/nfs
3. kubectl create secret generic edgefs-nfs-csi-driver-config --from-file=./edgefs-nfs-csi-driver-config.yaml
```

Deploy EdgeFS CSI NFS driver

After secret is created successfully, deploy EdgeFS CSI plugin, provisioner and attacher using the following command

```
1. cd cluster/examples/kubernetes/edgefs/csi/nfs
2. kubectl apply -f edgefs-nfs-csi-driver.yaml
```

There should be number of EdgeFS CSI plugin PODs available running as a DaemonSet:

```
1. ...
2. NAMESPACE     NAME                          READY   STATUS    RESTARTS   AGE
```

```
default       edgefs-nfs-csi-controller-0    4/4     Running   0          33s
default       edgefs-nfs-csi-node-9st9n      2/2     Running   0          33s
default       edgefs-nfs-csi-node-js7jp      2/2     Running   0          33s
default       edgefs-nfs-csi-node-lhjgr      2/2     Running   0          33s
...
```

At this point configuration is all ready and available for consumption by applications.

Pre-provisioned volumes (NFS)

This method allows to use already created exports in EdgeFS services. This method keeps exports provisioned after application PODs terminated. Read more on how to create PersistentVolume specification for pre-provisioned volumes:

Link to Pre-provisioned volumes manifest specification

To test creation and mount pre-provisioned volume to pod execute example

> **NOTE**: Make sure that `volumeHandle: segment:service@cluster/tenant/bucket` in nginx.yaml already exist on EdgeFS cluster and served via any Edgefs NFS service. Any volumeHandle's parameters may be omitted and will be substituted via CSI configuration file parameters.

Examples:

```
cd cluster/examples/kubernetes/edgefs/csi/nfs/examples
kubectl apply -f ./preprovisioned-edgefs-volume-nginx.yaml
```

Dynamically provisioned volumes (NFS)

To setup the system for dynamic provisioning, administrator needs to setup a StorageClass pointing to the CSI driver's external-provisioner and specifying any parameters required by the driver

Link to dynamically provisioned volumes specification

Note

For dynamically provisioned volumes kubernetes will generate volume name automatically (for example pvc-871068ed-8b5d-11e8-9dae-005056b37cb2) Additional creation options should be passed as parameters in StorageClass definition i.e :

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: edgefs-nfs-csi-storageclass
provisioner: io.edgefs.csi.nfs
parameters:
```

```
  segment: rook-edgefs
  service: nfs01
  tenant: ten1
  encryption: true
```

Parameters

> **NOTE**: Parameters and their options are case sensitive and should be in lower case.

Name	Description	Allowed values	Default value
`segment`	Edgefs cluster namespace for current StorageClass or PV.		rook-edgefs
`service`	Edgefs cluster service if not defined in secret		
`cluster`	Edgefs cluster namespace if not defined in secret		
`tenant`	Edgefs tenant namespace if not defined in secret		
`chunksize`	Chunk size for actual volume, in bytes	should be power of two	16384 bytes
`blocksize`	Block size for actual volume, in bytes	should be power of two	4096 bytes
`acl`	Volume acl restrictions		all
`ec`	Enables ccow erasure coding for volume	`true` , `false` , `0` , `1`	`false`
`ecmode`	Set ccow erasure mode data mode (If ‘ec’ option enabled)	`3:1:xor` , `2:2:rs` , `4:2:rs` , `6:2:rs` , `9:3:rs`	`6:2:rs`
`encryption`	Enables encryption for volume	`true` , `false` , `0` , `1`	`false`

Example:

```
cd cluster/examples/kubernetes/edgefs/csi/nfs/examples
kubectl apply -f ./dynamic-nginx.yaml
```

Apply Edgefs CSI ISCSI driver configuration

Check configuration options and create kubernetes secret for Edgefs CSI ISCSI plugin

```
cd cluster/examples/kubernetes/edgefs/csi/iscsi
kubectl create secret generic edgefs-iscsi-csi-driver-config --from-file=./edgefs-iscsi-csi-driver-config.yaml
```

Deploy Edgefs CSI ISCSI driver

After secret is created successfully, deploy EdgeFS CSI plugin, provisioner, attacher

and snapshotter using the following command

```
cd cluster/examples/kubernetes/edgefs/csi/iscsi
kubectl apply -f edgefs-iscsi-csi-driver.yaml
```

There should be number of EdgeFS CSI ISCSI plugin PODs available running as a DaemonSet:

```
...
NAMESPACE     NAME                              READY   STATUS    RESTARTS   AGE
default       edgefs-iscsi-csi-controller-0     4/4     Running   0          12s
default       edgefs-iscsi-csi-node-26464       2/2     Running   0          12s
default       edgefs-iscsi-csi-node-p5r58       2/2     Running   0          12s
default       edgefs-iscsi-csi-node-ptn2m       2/2     Running   0          12s
...
```

At this point configuration is all ready and available for consumption by applications.

Pre-provisioned volumes (ISCSI)

This method allows to use already created exports in EdgeFS ISCSI services. This method keeps exports provisioned after application PODs terminated. Read more on how to create PersistentVolume specification for pre-provisioned volumes:

Link to Pre-provisioned volumes manifest specification

To test creation and mount pre-provisioned volume to pod execute example:

> **NOTE**: Make sure that `volumeHandle: segment:service@cluster/tenant/bucket/lun` in nginx.yaml already exist on EdgeFS cluster and served via any Edgefs ISCSI service. Any volumeHandle's parameters may be omitted and will be substituted via CSI configuration file parameters.

Example:

```
cd cluster/examples/kubernetes/edgefs/csi/iscsi/examples
kubectl apply -f ./preprovisioned-edgefs-volume-nginx.yaml
```

Dynamically provisioned volumes (ISCSI)

For dynamic volume provisioning, the administrator needs to set up a *StorageClass* pointing to the driver. In this case Kubernetes generates volume name automatically (for example `pvc-ns-cfc67950-fe3c-11e8-a3ca-005056b857f8`). Default driver configuration may be overwritten in `parameters` section:

Link to dynamically provisioned volumes specification

Note for dynamically provisioned volumes

For dynamically provisioned volumes, Kubernetes will generate volume names automatically (for example pvc-871068ed-8b5d-11e8-9dae-005056b37cb2). To pass additional creation parameters, you can add them as parameters to your StorageClass definition.

Example:

```
1.  apiVersion: storage.k8s.io/v1
2.  kind: StorageClass
3.  metadata:
4.    name: edgefs-iscsi-csi-storageclass
5.  provisioner: io.edgefs.csi.nfs
6.  parameters:
7.    segment: rook-edgefs
8.    service: iscsi01
9.    cluster: cltest
10.   tenant: test
11.   bucket: bk1
12.   encryption: true
```

Parameters

> **NOTE**: Parameters and their options are case sensitive and should be in lower case.

Name	Description	Allowed values	Default value
segment	Edgefs cluster namespace for specific StorageClass or PV.		rook-edgefs
service	Edgefs cluster service if not defined in secret		
cluster	Edgefs cluster namespace if not defined in secret		
tenant	Edgefs tenant namespace if not defined in secret		
bucket	Edgefs bucket namespace if not defined in secret		
chunksize	Chunk size for actual volume, in bytes	should be power of two	16384
blocksize	Blocksize size for actual volume, in bytes	should be power of two	4096
fsType	New volume’s filesystem type	ext4 , ext3 , xfs	ext4
acl	Volume acl restrictions		all
ec	Enables ccow erasure coding for volume	true , false , 0 , 1	false
ecmode	Set ccow erasure mode data mode (If ‘ec’ option enabled)	4:2:rs , 6:2:rs , 4:2:rs , 9:3:rs	4:2:rs

`encryption`	Enables encryption for volume	`true` , `false` , `0` , `1`	`false`

Example:

```
cd cluster/examples/kubernetes/edgefs/csi/nfs/examples
kubectl apply -f ./dynamic-nginx.yaml
```

EdgeFS CSI ISCSI driver snapshots and clones

Getting information about existing snapshots

```
# snapshot classes
kubectl get volumesnapshotclasses.snapshot.storage.k8s.io

# snapshot list
kubectl get volumesnapshots.snapshot.storage.k8s.io

# volumesnapshotcontents
kubectl get volumesnapshotcontents.snapshot.storage.k8s.io
```

To create volume’s clone from existing snapshot you should:

- Create snapshotter StorageClass Example yaml
- Have an existing PVC based on EdgeFS ISCSI LUN
- Take snapshot from volume Example yaml
- Clone volume from existing snapshot Example yaml

Troubleshooting and log collection

For details about other configuration and deployment of NFS, ISCSI and EdgeFS CSI plugin, see Wiki pages:

- Quick Start Guide

Please submit an issue at: Issues

Troubleshooting

- Show installed drivers:

```
kubectl get csidrivers.csi.storage.k8s.io
kubectl describe csidrivers.csi.storage.k8s.io
```

- Error:

```
MountVolume.MountDevice failed for volume "pvc-ns-<...>" :
driver name io.edgefs.csi.iscsi not found in the list of registered CSI drivers
```

Make sure *kubelet* is configured with `--root-dir=/var/lib/kubelet`, otherwise update paths in the driver yaml file (all requirements).

- “VolumeSnapshotDataSource” feature gate is disabled:

```
vim /var/lib/kubelet/config.yaml
# ...
# featureGates:
#   VolumeSnapshotDataSource: true
# ...
vim /etc/kubernetes/manifests/kube-apiserver.yaml
# ...
#     - --feature-gates=VolumeSnapshotDataSource=true
# ...
```

- Driver logs (for ISCSI driver, to get NFS driver logs substitute iscsi to nfs)

```
kubectl logs -f edgefs-iscsi-csi-controller-0 driver
kubectl logs -f $(kubectl get pods | awk '/edgefs-iscsi-csi-node-/ {print $1;exit}') driver
# combine all pods:
kubectl get pods | awk '/edgefs-iscsi-csi-node-/ {system("kubectl logs " $1 " driver &")}'
```

- Show termination message in case driver failed to run:

```
kubectl get pod edgefs-iscsi-csi-controller-0 -o go-template="{{range .status.containerStatuses}}
{{.lastState.terminated.message}}{{end}}"
```

EdgeFS Monitoring

Each Rook EdgeFS cluster has some built in metrics collectors/exporters for monitoring with Prometheus. If you do not have Prometheus running, follow the steps below to enable monitoring of Rook. If your cluster already contains a Prometheus instance, it will automatically discover Rooks scrape endpoint using the standard `prometheus.io/scrape` and `prometheus.io/port` annotations.

Prometheus Operator

First the Prometheus operator needs to be started in the cluster so it can watch for our requests to start monitoring Rook and respond by deploying the correct Prometheus pods and configuration. A full explanation can be found in the Prometheus operator repository on GitHub, but the quick instructions can be found here:

```
1. kubectl apply -f https://raw.githubusercontent.com/coreos/prometheus-operator/v0.27.0/bundle.yaml
```

This will start the Prometheus operator, but before moving on, wait until the operator is in the `Running` state:

```
1. kubectl get pod
```

Once the Prometheus operator is in the `Running` state, proceed to the next section.

Prometheus Instances

With the Prometheus operator running, we can create a service monitor that will watch the Rook cluster and collect metrics regularly. From the root of your locally cloned Rook repo, go the monitoring directory:

```
1. git clone --single-branch --branch release-1.4 https://github.com/rook/rook.git
2. cd rook/cluster/examples/kubernetes/edgefs/monitoring
```

Create the service monitor as well as the Prometheus server pod and service:

```
1. kubectl create -f service-monitor.yaml
2. kubectl create -f prometheus.yaml
3. kubectl create -f prometheus-service.yaml
```

Ensure that the Prometheus server pod gets created and advances to the `Running` state before moving on:

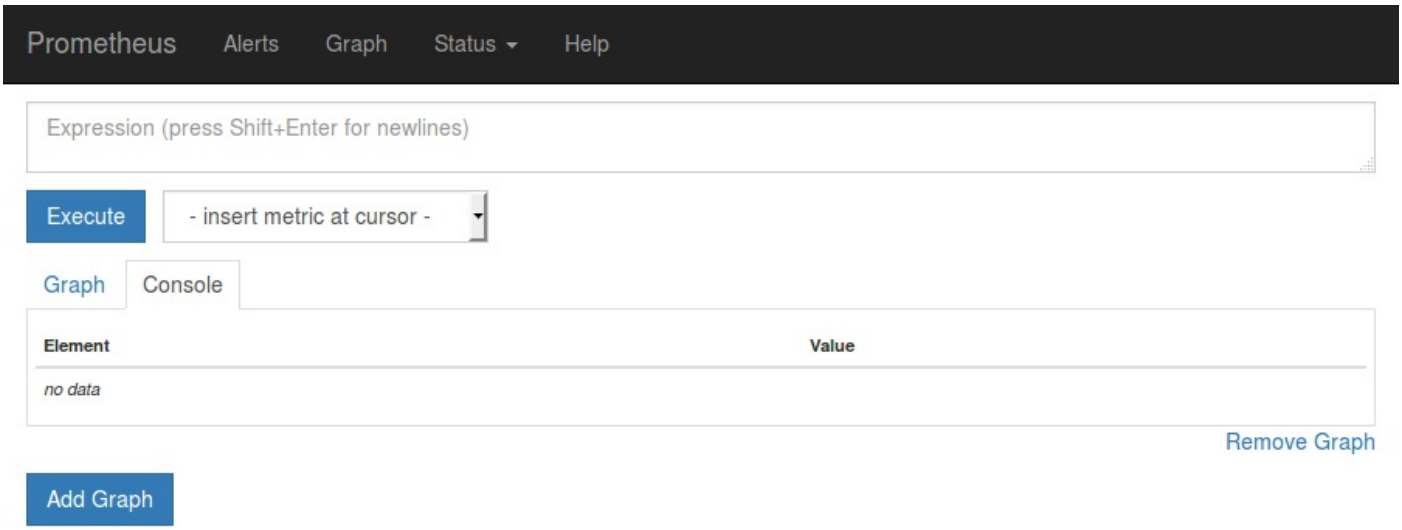
```
1. kubectl -n rook-edgefs get pod prometheus-rook-prometheus-0
```

Prometheus Web Console

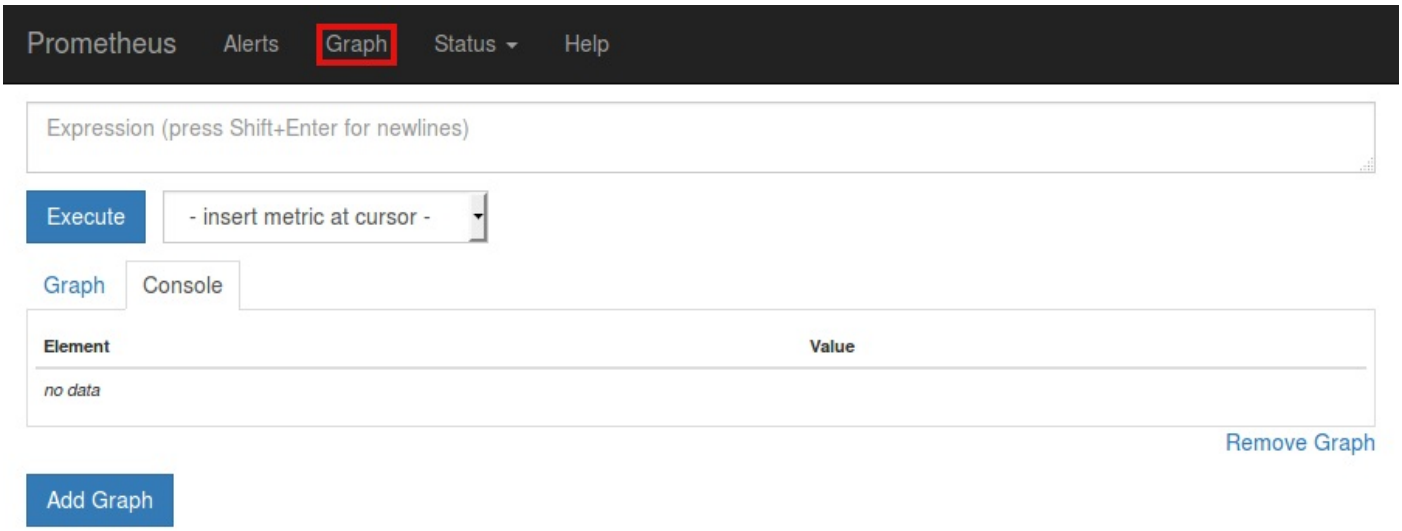
Once the Prometheus server is running, you can open a web browser and go to the URL that is output from this command:

```
echo "http://$(kubectl -n rook-edgefs -o jsonpath={.status.hostIP} get pod prometheus-rook-prometheus-0):30900"
```

You should now see the Prometheus monitoring website.



Click on `Graph` in the top navigation bar.



In the dropdown that says `insert metric at cursor`, select any metric you would like to see, for example `nedge_cluster_objects`. Below the `Execute` button, ensure the `Graph` tab is selected and you should now see a graph of your chosen metric over time.

Prometheus Consoles

A guide to how you can write your own Prometheus consoles can be found on the official Prometheus site here: Prometheus.io Documentation - Console Templates.

Grafana Dashboards

For feedback on the dashboards please reach out to him on the Rook.io Slack.

> **NOTE**: The dashboard only compatible with Grafana 5.0.3 or higher.

The following Grafana dashboard is available:

- EdgeFS Cluster

Teardown

To clean up all the artifacts created by the monitoring walkthrough, copy/paste the entire block below (note that errors about resources “not found” can be ignored):

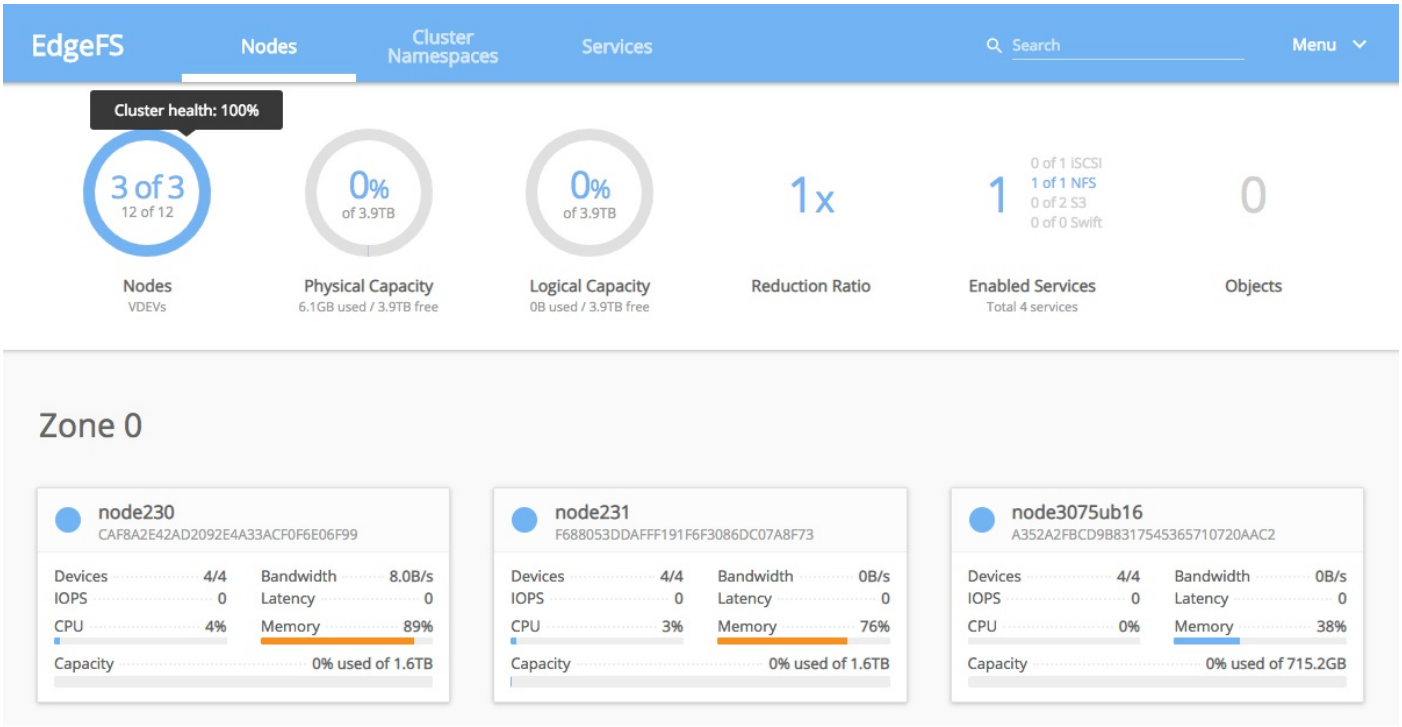
```
kubectl delete -f service-monitor.yaml
kubectl delete -f prometheus.yaml
kubectl delete -f prometheus-service.yaml
kubectl delete -f https://raw.githubusercontent.com/coreos/prometheus-operator/v0.27.0/bundle.yaml
```

Then the rest of the instructions in the Prometheus Operator docs can be followed to finish cleaning up.

EdgeFS Dashboard and User Interface

EdgeFS comes with built-in local cluster Dashboard and User Interface.

The dashboard is a very helpful tool to give you an overview of the status of your cluster, including overall health, status of the corosync quorum, status of the VDEVs, and other EdgeFS services.



Rook EdgeFS Operator enables the dashboard by default. A K8s service will be created to expose HTTP/S ports inside the cluster.

This example shows what ports was configured for EdgeFS right after successful cluster deployment:

```
1. # kubectl get svc -n rook-edgefs
   NAME                  TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
2. AGE
   rook-edgefs-restapi   NodePort    10.105.177.93   <none>        8881:32135/TCP,8080:31363/TCP,4443:31485/TCP
3. 3h14m
   rook-edgefs-ui        NodePort    10.97.70.209    <none>        3000:32048/TCP,3443:30273/TCP
4. 3h14m
   rook-edgefs-mgr       NodePort    10.108.199.1    <none>        6789:31370/TCP
5. 3h14m
```

The `rook-edgefs-restapi` service is used for the Prometheus metrics scrape and REST API endpoints. REST API endpoint is used by Console and Graphical UIs. It exposing ClusterIPs with HTTP ports 8080 and 4443(SSL). NodePort(s) for external access automatically pre-created.

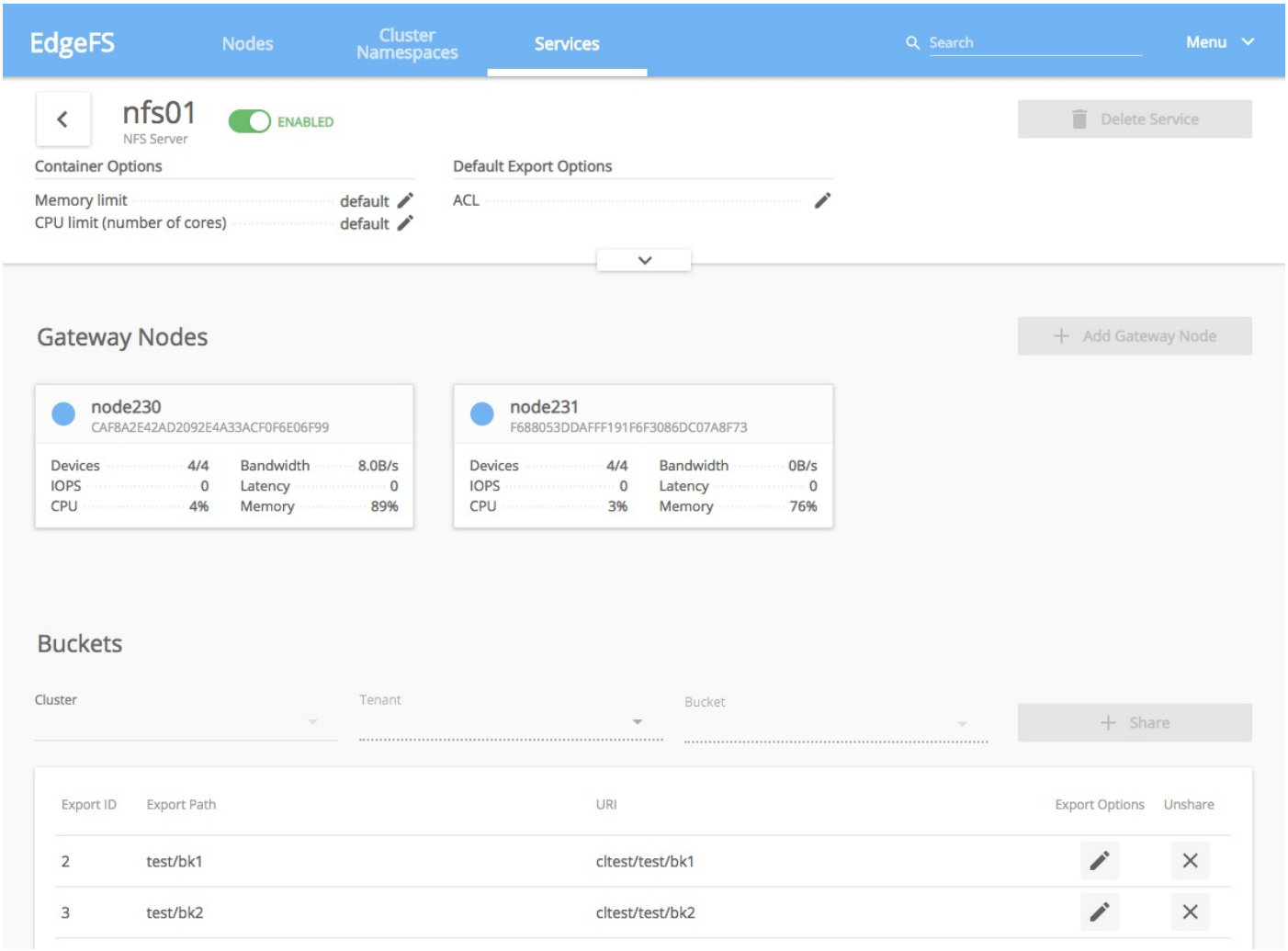
The `rook-edgefs-ui` service is used to expose browser based User Interface ports. Using

NodePort you will be able to connect to the dashboard by direct node IP and provided port number, in this example at `http://NODEIP:32048` .

The `rook-edgefs-mgr` service is for gRPC communication between the EdgeFS CSI Provisioner, REST API and services. It exposes cluster internal port 6789.

Dynamic EdgeFS Service CRD editing

One of the very useful features of EdgeFS Dashboard is Dynamic Service CRD editing UI. Ability to edit CRDs via intuitive graphical interface can simplify on-going EdgeFS cluster management drastically. For instance, below picture demonstrates how NFS CRD service can be dynamically edited, exports can be added or removed:



EdgeFS UI renders user requests and sends them to the REST API where service CRD is constructed, validated and executed. Certain parameters can only be updated within a cluster service definition itself, and as such this UI blends the two, thus providing simplified management functionality.

Credentials

After you connect to the dashboard you will need to login for secure access. Rook

EdgeFS operator creates a default user named `admin` and password `edgefs`. It can be changed later via `neadm system passwd` command executed within `mgr` pod.

EdgeFS VDEV Management

EdgeFS can run on top of any block device. It can be a raw physical or virtual disk (RT-RD type). Or it can be a directory on a local filesystem (RT-LFS type). In case of a local filesystem, VDEV function will be emulated via memory-mapped files and as such, management of the files is a responsibility of underlying filesystem of choice. (e.g. ext4, xfs, zfs, etc). This document describes the management of VDEVs built on top of raw disks (RT-RD type).

EdgeFS on-disk organization

EdgeFS converts underlying block devices (VDEVs) into a local high-performance, memory and SSD/NVMe optimized key-value databases.

Persistent data

The EdgeFS chunks a data object into one or several data chunk which are members of the `TT_CHUNK_PAYLOAD` data type that forms the first data type group: *the persistent data*. This group also includes a configuration data type called `TT_HASHCOUNT` .

Persistent metadata

To define an object assembly order there are two manifest metadata types: `TT_CHUNK_MANIFEST` and `TT_VERSION_MANIFEST` . If an object is EC-protected, then an additional entry of `TT_PARTIY_MANIFEST` metadata type will be added to each leaf manifest. Payload's and manifest's lifespan depends on the presence of a tiny object of `TT_VERIFIED_BACKREF` type that tracks de-duplication back references in background operations. All the objects are indexed by its name within a cluster namespace and the index is stored in a `TT_NAMEINDEX` metadata type. Types `TT_CHUNK_MANIFEST` , `TT_VERSION_MANIFEST` , `TT_PARTIY_MANIFEST` , `TT_VERIFIED_BACKREF` and `TT_NAMEINDEX` form the second group: *the persistent metadata*.

Temporary metadata

And the third group is *the temporary metadata*. It includes on-disk data queues whose entries are removed after being processed by server's background jobs: `TT_VERIFICATION_QUEUE` , `TT_BATCH_QUEUE` , `TT_INCOMING_BATCH_QUEUE` , `TT_ENCODING_QUEUE` , `TT_REPLICATION_QUEUE` and `TT_TRANSACTION_LOG` .

Persistent data	Persistent metadata	Temporary metadata

`TT_CHUNK_PAYLOAD`
`TT_HASHCOUNT`
	`TT_CHUNK_MANIFEST`
	`TT_VERSION_MANIFEST`
	`TT_PARTIY_MANIFEST`
	`TT_VERIFIED_BACKREF`
	`TT_NAMEINDEX`
		`TT_VERIFICATION_QUEUE`
		`TT_BATCH_QUEUE`
		`TT_INCOMING_BATCH_QUEUE`
		`TT_ENCODING_QUEUE`
		`TT_REPLICATION_QUEUE`
		`TT_TRANSACTION_LOG`

Consistency considerations

Each VDEV in the cluster holds a certain number of entries of each data type. Damage of data table of different groups has a different impact on VDEV's consistency. Usually, if a damaged data type belongs to *the temporary metadata*, then such data table can be dropped without a noticeable influence on the cluster. If a persistent data or persistent metadata gets corrupted, then impact will take place, however, it will never be vital for the cluster thanks to data protection approaches the EdgeFS implements: data redundancy or erasure coding. Moreover, the underlying device management layer also tries to reduce the cost of data loss by means of different techniques, like data sharding.

> **IMPORTANT**: In EdgeFS design, loss of a data or metadata chunk generally do not affect front I/O performance due to unique dynamic placement and retrieval technique. Data placement or retrieval gets negotiated prior to each chunk I/O. This allows EdgeFS to select the most optimal location for I/O rather then hard-coded. With this, failing device(s) or temporary disconnected/busy devices will not affect overall local cluster performance.

Stronger consistency/durability comes with a performance price and it highly depends on an use cases. EdgeFS provides needed flexibility of selecting most optimal consistency model via usage of `sync` parameter that defines default behavior of write operations at device or directory level. Acceptable values are 0, 1 (default), 2, 3:

- `0`: No syncing will happen. Highest performance possible and good for HPC scratch types of deployments. This option will still sustain crash of pods or software bugs. It will not sustain server power loss an may cause node/device level inconsistency.
- `1`: Default method. Will guarantee node / device consistency in case of power loss with reduced durability.
- `2`: Provides better durability in case of power loss at the cost of extra metadata syncing.
- `3`: Most durable and reliable option at the cost of significant performance impact.

EdgeFS RT-RD Architecture

The main metrics of the RT-RD device is a partition level (the *plevel*). The *plevel* defines a number of disk partitions user's data will be split across: for each key-value pair, the key identifies the *plevel index*. The RT-RD splits entire raw HDD (or SSD) into several partitions. The following partition types are defined:

- Main partitions. Resides on the HDD and used as the main data store. A number of partitions equal to disk *plevel*. In a all SSD or all HDD configurations, main partitions keep all data types.
- Write-ahead-log (*WAL*) partitions. The *WAL* drastically improves write performance, especially for a hybrid (HDDs+SSD) configuration. There is one *WAL* partition per *plevel* Its location depends on configuration type. In a hybrid configuration, it's on SSD, otherwise on the same disk as the corresponding *main* partition.

- A metadata offload partition (*mdoffload*). For the hybrid configuration-only. One partition per an HDD. It is used to store temporary metadata, indexes and some persistent metadata types (optionally).
- A configuration partition. A tiny partition per device which keeps a disk configuration options (a *metaloc entry*).

Regardless of location, each partition keeps an instance of a key-value database (*the environment*) and a number sub-databases for different data types within the environment. A sub-database is referenced by its Data Base Identifier(*DBI*) which contains data type name as a part of the *DBI* and depends on the environment's location. Along with 3 aforementioned data type categories, the RT-RD keeps an extra one for a hybrid device - the *mdoffload index*. The *mdoffload index* is used to improve the performance of GET operations that target the data types on HDD without data retrieval: chunk stat or an attribute get. A damaged *mdoffload index* can always be restored from data on HDD.

Below is a data type to DBI mapping table

Group	Data type	DBI	Location
Persistent data	TT_CHUNK_PAYLOD	bd-part-TT_CHUNK_PAYLOAD-0	HDD
	TT_HASHCOUNT	bd-part-TT_HASHCOUNT-0	HDD
Persistent metadata	TT_CHUNK_MANIFEST	TT_CHUNK_MANIFEST bd-part-TT_CHUNK_MANIFEST-0	SSD HDD
	TT_VERSION_MANIFEST	TT_VERSION_MANIFEST bd-part-TT_VERSION_MANIFEST-0	SSD HDD
	TT_PARITY_MANIFEST	TT_PARITY_MANIFEST bd-part-TT_PARITY_MANIFEST-0	SSD HDD
	TT_NAMEINDEX	TT_NAMEINDEX bd-part-TT_NAMEINDEX-0	SSD HDD
	TT_VERIFIED_BACKREF	TT_VERIFIED_BACKREF bd-part-TT_VERIFIED_BACKREF-0	SSD HDD
Temporary metadata	TT_VERIFICATION_QUEUE	TT_VERIFICATION_QUEUE bd-part-TT_VERIFICATION_QUEUE-0	SSD HDD
	TT_BATCH_QUEUE	TT_BATCH_QUEUE bd-part-TT_BATCH_QUEUE-0	SSD HDD
	TT_INCOMING_BATCH_QUEUE	TT_INCOMING_BATCH_QUEUE bd-part-TT_INCOMING_BATCH_QUEUE-0	SSD HDD
	TT_ENCODING_QUEUE	TT_ENCODING_QUEUE bd-part-TT_ENCODING_QUEUE-0	SSD HDD
	TT_REPLICATION_QUEUE	TT_REPLICATION_QUEUE bd-part-TT_REPLICATION_QUEUE-0	SSD HDD
	TT_TRANSACTION_LOG	TT_TRANSACTION_LOG bd-part-TT_TRANSACTION_LOG-0	SSD HDD
Mdoffload index	-	keys-TT_CHUNK_MANIFEST keys-TT_CHUNK_PAYLOAD keys-TT_VERSION_MANIFEST	SSD SSD SSD
	-	mdcache-TT_CHUNK_MANIFEST	SSD

	-	mdcache-TT_VERSION_MANIFEST	SSD

So now you are informed enough to understand next paragraphs.

Cluster Health Verification

Login to the toolbox as shown in this example:

```
    kubectl exec -it -n rook-edgefs 'kubectl get po --all-namespaces | awk '{print($2)}' | grep edgefs-mgr' -- env
1.  COLUMNS=$COLUMNS LINES=$LINES TERM=linux toolbox
```

To find out what device needs to go into the maintenance state, run the following command:

```
1. # efscli system status -v1
2. ServerID CFAE1E62652370A93769378B2C862F23 ubuntu1632243:rook-edgefs-target-0 DEGRADED
3. VDEVID D76242575CAA2662862CEEBC65D0B69F ata-ST1000NX0423_W470M48T ONLINE
4. VDEVID B2BB69729BC3EB9ECE5F0DCB3DB4D0D6 ata-ST1000NX0423_W470NQ7A FAULTED
5. VDEVID BB287F0C6747B1E59A85872B2C7F39B3 ata-ST1000NX0423_W470P9XJ ONLINE
6. VDEVID 82F9A02F0D2A1BC44E6AF8D2B455189D ata-ST1000NX0423_W470M3JK ONLINE
7. ServerID 1A0B10BCF8CB0E6D34451B4D3F84CE97 ubuntu1632240:rook-edgefs-target-1 ONLINE
8. VDEVID 904ED942BD62FF4A997C4A23E2B8043B ata-ST1000NX0423_W470NLFR ONLINE
9. ...
```

From this command you will see that pod ‘rook-edgefs-target-0’ is in degraded state and device ‘ata-ST1000NX0423_W470NQ7A’ needs maintenance.

VDEV Verification

Login to the affected target pod as shown in this example:

```
1. kubectl exec -it -n rook-edgefs rook-edgefs-target-0 -- env COLUMNS=$COLUMNS LINES=$LINES TERM=linux toolbox
```

Make sure the disk is faulted:

```
1. # efscli device list
2.
3. NAME                        | VDEV ID                          | PATH     | STATUS
4. +---------------------------+----------------------------------+----------+--------+
5. ata-ST1000NX0423_W470M3JK   | 82F9A02F0D2A1BC44E6AF8D2B455189D | /dev/sdb | ONLINE
6. ata-ST1000NX0423_W470M48T   | D76242575CAA2662862CEEBC65D0B69F | /dev/sdc | ONLINE
7. ata-ST1000NX0423_W470P9XJ   | BB287F0C6747B1E59A85872B2C7F39B3 | /dev/sdd | ONLINE
8. ata-ST1000NX0423_W470NQ7A   | B2BB69729BC3EB9ECE5F0DCB3DB4D0D6 | /dev/sde | UNAVAILABLE
```

Use `efscli device detach ata-ST1000NX0423_W470NQ7A` for detaching a disk. It will be marked a faulted and won’t be attached at the next `ccow-daemon` restart. Also, the preserved

detached state can be cleared by a command `nezap --disk=diskID --restore-metaloc` .

Once detached, related HDD/SSD partitions can be inspected/fixed by means of `efscli device check` command or zapped. When maintenance is done, the VDEV(s) can become operational again by invoking a command `efscli device attach` .

Assuming that your disk is detached and your verification procedure can be initiated. The tool is accessible via `efscli device check` command. It provides an interactive user interface for device validation and recovery. Before getting started, a user needs to define a scratch area location. It will be used as a temporary store for environments being compacted or recovered. The scratch area can be defined in terms of data path within filesystem, a path to a raw disk (or its partition) in the `/dev/` folder or raw disk/partition ID. User has to make sure the filesystem can provide enough free space to keep data from single `plevel` . The same requirement is for raw disk/partition size: at least 600GB, 1TB is recommended. A user can define the scratch area path in `$(NEDGE_HOME)/etc/ccow/rt-rd.json` as follow:

```
{
"devices": [...],
"scratch": "/tmp.db"
}
```

Alternatively, the path can be specified by `-s <path>` flag. For example: `efscli device check -s /scratch.db ata-ST1000NX0423_W470NQ7A`

The following actions will be completely interactive and a user's confirmation will be required before any important disk changes. The command output might look like follow. We split it into parts for convenience

```
$ efscli device check -s /scratch.db ata-ST1000NX0423_W470NQ7A
INFO: checking disk /dev/sde. Stored metaloc record info:

NAME      | TYPE | ID                                | PATH       | CAPACITY | USED | PSIZE | BCACHE | STATUS
+---------+------+-----------------------------------+------------+----------+------+-------+--------+----------
-+
PLEVEL1   | Main | ata-ST1000NX0423_W470NQ7A-part1 | /dev/sde1  | 465.76G  | 0%   | 32k   | OFF    | None
          | WAL  | scsi-35000c5003021f02f-part7    | /dev/sdf7  | 2.67G    |      | 4k    | n/a    | None
PLEVEL2   | Main | ata-ST1000NX0423_W470NQ7A-part2 | /dev/sde2  | 465.76G  | 0%   | 32k   | OFF    | None
          | WAL  | scsi-35000c5003021f02f-part8    | /dev/sdf8  | 2.67G    |      | 4k    | n/a    | None
OFFLOAD   |      | scsi-35000c5003021f02f-part12   | /dev/sdf12 | 180.97G  | 0%   | 8k    | n/a    | CORRUPTED
```

The first table displays the VDEV configuration and its known errors (if any). In our example the VDEV is a hybrid one (The OFFLOAD partition signals that) with 2 *plevels* situated on the HDD `/dev/sde` . Main partitions are `/dev/sde1` and `/dev/sde2` , WAL partitions are on SSD (`/dev/sdf7` and `/dev/sdf8`) as well as the mdoffload partition `/dev/sdf12` . For each partition we see its capacity, utilization, logical paze size (internal for the key-value engine), bcache presence and healthy status. `None` means the partition doesn't have know errors. However, the `/dev/sdf12` is marked as a `CORRUPTED` and will be check by the verification algorithm. If there known errors are

absent, then user will be asked for extended data validation of each partition. It's important to mention, that if a VDEV is online and user wants to validate it, then such VDEV will be set `READ-ONLY` until validation is in progress. If there are any errors, then the VDEV will be detached for maintenance. However, in our case the problem is known and it requires further validation in order to find a proper recovery solution.

```
WARN: a fault record is detected for mdoffload (/dev/sdf12)
INFO: disk ata-ST1000NX0423_W470NQ7A is UNAVAILABLE
INFO: locking the device
INFO: 1 partition(s) needs to be validated
INFO: validating /dev/sdf12
Progress: [==============================>] 100%
DBI NAME                    | ENTRIES | OPEN TEST | READ TEST          | CORRUPTED | WRITE TEST
+--------------------------+---------+-----------+--------------------+-----------+------------+
TT_TRANSACTION_LOG          | 0       | PASSED    | PASSED             | N/A       | SKIPPED
keys-TT_VERSION_MANIFEST    | 28999   | PASSED    | PASSED             | N/A       | SKIPPED
TT_BATCH_INCOMING_QUEUE     | 0       | PASSED    | PASSED             | N/A       | SKIPPED
TT_ENCODING_QUEUE           | 0       | PASSED    | PASSED             | N/A       | SKIPPED
TT_PARITY_MANIFEST          | 0       | PASSED    | PASSED             | N/A       | SKIPPED
TT_REPLICATION_QUEUE        | 0       | PASSED    | PASSED             | N/A       | SKIPPED
keys-TT_CHUNK_MANIFEST      | 56448   | PASSED    | PASSED             | N/A       | SKIPPED
TT_BATCH_QUEUE              | 0       | PASSED    | PASSED             | N/A       | SKIPPED
TT_VERIFIED_BACKREF         | 55403   | PASSED    | PASSED             | N/A       | SKIPPED
keys-TT_CHUNK_PAYLOAD       | 212787  | PASSED    | KEY FORMAT ERROR   | N/A       | N/A
TT_CHUNK_MANIFEST           | 56448   | PASSED    | CORRUPTED          | N/A       | N/A
TT_NAMEINDEX                | 189     | PASSED    | PASSED             | N/A       | SKIPPED
TT_VERIFICATION_QUEUE       | 29      | PASSED    | PASSED             | N/A       | SKIPPED
TT_VERSION_MANIFEST         | 28999   | PASSED    | DB STRUCTURE ERROR | N/A       | N/A

ERROR: the mdoffload environment got unrecoverable damages.
ENTIRE device /dev/sdf12 needs to be formatted. All the data will be lost.
Press 'Y' to start [y/n]: y
```

The validation is performed on a sub-database (DB) basis. For each DB there are up to 3 tests: open, read and modify. The last (modify) is disabled by default but can be activated in a policy file. The open and read tests are able to discover an overwhelming majority of structural errors without detaching a disk from ccow-daemon: if an environment is online, the user will be asked for permission to switch the device to read-only mode before any tests. The modify test requires the device to be set unavailable. Validation result is shown as a table for each DBI.

The table has a column named `CORRUPTED` which may show number key-value pairs whose value's hash ID doesn't match expected one. For certain data types, it's acceptable to have a limited number of damaged entries. It's a compromise between data lost cost and data integrity. Often we don't want to mark the whole DB as faulted due to just a few damaged values which will be detected and removed by ccow-daemon soon or later.

Further behaviour depends on validation results:

- If there are no corrupted DB(s), the environment is considered healthy.
- If there are damaged DB(s), then the behaviour depends on DB's error handling policy. By default it implies the following:
- If corrupted only temporary metadata or those data can be reconstructed (the *mdoffload index*), then a selective recovery will be suggested to the user. The selective recovery makes copies of non-damaged DBs only. Corrupted ones will be re-created or re-constructed when the device is attached.
- Corrupted data or metadata table with sensitive data. In this case, a *plevel* or whole device needs to be formatted. All damaged environments will be added to a format queue that will be processed on the final stage.

As we can see 3 DBs are corrupted. One of them (`keys-TT_CHUNK_PAYLOAD`) can be easily recovered (the *mdoffload index*), however `TT_NAMEINDEX` and `TT_VERSION_MANIFEST` cannot be reconstructed and they reside on a mdoffload partition. So the whole device needs to be formatted.

```
1. WARN: the entire device is about to be formatted.
2. ALL the data on it will be LOST. Do you want to proceed? [y/n]: y
3. INFO: formatting entire device ata-ST1000NX0423_W470NQ7A
4. INFO: format done
5.
6. INFO: device examination summary:
7. VALIDATED    | FORMATTED  | RECOVERED | COMPACTIFIED
8. +------------+------------+-----------+--------------+
9. /dev/sdf12   | /dev/sde1  |           |
10.             | /dev/sde2  |           |
11.             | /dev/sdf12 |           |
```

A user confirmed and the VDEV has been formatted. Check is done. The VDEV can be put online by a command `efscli device attach ata-ST1000NX0423_W470NQ7A`

VDEV Replacement Procedure

A command `efscli disk replace [-f] [-y] <old-name> <new-name>` performs on-the-fly disk substitution. The disk with name `old-name` will be detached and disk with name `new-name` will be configured and used instead of the old one. If the `new-disk` has a partition table on it, then the command will fail unless the `-f` flag is specified. When the flag set, the user will be asked for permission to destroy the partition table. The '-y' flags forces destruction of the partition table without confirmation. The `replace` command can be used when EdgeFS service is down. In this case, the new disk will be attached upon the next service start.

Login to the affected target pod where disk needs to be replaced as shown in this example:

```
kubectl exec -it -n rook-edgefs rook-edgefs-target-0 -- env COLUMNS=$COLUMNS LINES=$LINES TERM=linux toolbox
```

List available and unused disks:

```
# efscli device list -s
NAME                        | VDEV ID                          | PATH     | STATUS
+---------------------------+----------------------------------+----------+-------------+
ata-ST1000NX0423_W470M4BZ   |                                  | /dev/sda | UNUSED
ata-ST1000NX0423_W470M3JK   | 147A81246937AEFF934D00C8DB92C4D3 | /dev/sdb | ONLINE
ata-ST1000NX0423_W470M48T   | DCA10919B6F55A66A23BC5916642DD7E | /dev/sdc | ONLINE
ata-ST1000NX0423_W470P9XJ   | C82B78AC845989DC731BF59FE705256A | /dev/sdd | ONLINE
ata-ST1000NX0423_W470NQ7A   | BCA4C8F096DE96B4B350DF4F92E30F19 | /dev/sde | UNAVAILABLE
```

There is a device with an `UNUSED` or `PARTITIONED` status. The disk `ST1000NX0423_W470M4BZ` is not used and can replace a faulted one.

```
# efscli device replace ata-ST1000NX0423_W470NQ7A ata-ST1000NX0423_W470M4BZ

INFO: Probbing disk ata-ST1000NX0423_W470M4BZ
INFO: Attaching disk ata-ST1000NX0423_W470M4BZ
INFO: The disk is replaced succefully
```

VDEV server recovery

If a database partition gets corrupted, then there is a small probability for the VDEV container to enter an infinite restart loop. In order to prevent a container from restart use the following command:

```
kubectl exec -it -n rook-edgefs rook-edgefs-target-<n> -c auditd -- touch /opt/nedge/var/run/.edgefs-start-block-ccowd
```

where `rook-edgefs rook-edgefs-target-<n>` is the affected pod ID.

Once command is done, you must be able to reach a toolbox console:

```
kubectl exec -it -n rook-edgefs rook-edgefs rook-edgefs-target-<n> -- env COLUMNS=$COLUMNS LINES=$LINES TERM=linux toolbox
```

and run the `efscli device check ...` command.

When you are done, before exiting the toolbox, do not forget to remove the file /opt/nedge/var/run/.edgefs-start-block-ccowd

EdgeFS Upgrades

This guide will walk you through the manual steps to upgrade the software in a Rook EdgeFS cluster from one version to the next. Rook EdgeFS is a multi-cloud distributed software system and therefore there are multiple components to individually upgrade in the sequence defined in this guide. After each component is upgraded, it is important to verify that the cluster returns to a healthy and fully functional state.

We welcome feedback and opening issues!

Supported Versions

The supported version for this upgrade guide is **from a 1.0 release to a 1.x releases**. Build-to-build upgrades are not guaranteed to work. This guide is to perform upgrades only between the official releases.

Upgrades from Alpha to Beta not supported. However, please see migration procedure below.

EdgeFS Migration

EdgeFS Operator provides a way of preserving data on disks or directories while moving to a new version (like Alpha to Beta transitioning) or reconfiguring (like full re-start).

We will do all our work in the EdgeFS example manifests directory.

```
cd $YOUR_ROOK_REPO/cluster/examples/kubernetes/edgefs/
```

Example of migration from existing `v1beta1` EdgeFS cluster to stable `v1` EdgeFS cluster: For already existing EdgeFS manifests we should replace `v1beta1` version to `v1`. We can use a few simple `sed` commands to do this for all manifests at once.

```
sed -i.bak -e "s/edgefs.rook.io\/v1beta1/edgefs.rook.io\/v1beta1/g" *.yaml
sed -i -e "s/edgefs.rook.io\/v1beta1/edgefs.rook.io\/v1/g" *.yaml
mkdir -p backups
mv *.bak backups/
```

Then we should add new `v1` version specification to existing EdgeFS' CRDs.

```
kubectl apply -f upgrade-from-v1beta1-create.yaml
```

And replace EdgeFS operator to new stable 'v1' version

```
kubectl -n rook-edgefs-system set image deploy/rook-edgefs-operator rook-edgefs-operator=rook/edgefs:v1.1.0
```

Then you could update your cluster's EdgeFS image for latest one as discribed below.

EdgeFS Version Upgrade

Official EddgeFS container images can be found on Docker Hub.

1. Update the EdgeFS image used for the EdgeFS cluster:

```
# Parameterize the environment
export ROOK_SYSTEM_NAMESPACE="rook-edgefs-system"
export CLUSTER_NAME="rook-edgefs"
```

The majority of the upgrade will be handled by the Rook operator. Begin the upgrade by changing the EdgeFS image field in the cluster CRD (`spec:edgefsImageName`).

```
NEW_EDGEFS_IMAGE='edgefs/edgefs:latest'
kubectl -n $CLUSTER_NAME patch Cluster $CLUSTER_NAME --type=merge \
  -p "{\"spec\": {\"edgefsImageName\": \"$NEW_EDGEFS_IMAGE\"}}"
```

or via console editor update the `edgefsImageName` property:

```
kubectl edit -n $CLUSTER_NAME Cluster $CLUSTER_NAME
```

and save results.

1. Wait for the pod updates to complete:

As with upgrading Rook, you must now wait for the upgrade to complete. Determining when the EdgeFS version has fully updated is rather simple.

```
kubectl -n $CLUSTER_NAME describe pods | grep "Image:" | sort | uniq
# This cluster is not yet finished:
#      Image:         edgefs/edgefs:1.2.31
#      Image:         edgefs/edgefs:1.2.50
#      Image:         edgefs/edgefs-restapi:1.2.31
#      Image:         edgefs/edgefs-ui:1.2.31
# This cluster is also finished(all versions are the same):
#      Image:         edgefs/edgefs:1.2.50
#      Image:         edgefs/edgefs-restapi:1.2.50
#      Image:         edgefs/edgefs-ui:1.2.50
```

1. Verify the updated EdgeFS cluster:

Access to EdgeFS mgr pod and check EdgeFS system status

```
kubectl exec -it -n $CLUSTER_NAME rook-edgefs-mgr-xxxx-xxx -- toolbox
efscli system status -v 1
```

EdgeFS Nodes Update

Nodes can be added and removed over time by updating the Cluster CRD, for example with `kubectl edit Cluster -n rook-edgefs`. This will bring up your default text editor and allow you to add and remove storage nodes from the cluster. This feature is only available when `useAllNodes` has been set to `false` and `resurrect` mode is not used.

Add node Example

1. Edit Cluster CRD `kubectl edit Cluster -n rook-edgefs`
2. Add new node section with desired configuration in storage section of Cluster CRD:

Currently we adding new node `node3072ub16` with two drives `sdb` and `sdc` on it.

```
    - config: null
      devices:
      - FullPath: ""
        config: null
        name: sdb
      - FullPath: ""
        config: null
        name: sdc
      name: node3072ub16
      resources: {}
```

- Save CRD and operator will update all target nodes and related pods of the EdgeFS cluster.
- Login to EdgeFS mgr toolbox and adjust FlexHash table to a new configuration using `efscli system fhtable` command.

EdgeFS Operator Upgrade

In terms of to bring new Rook Operator features on a running system, it needs to be upgraded first. And once upgraded, you will need to create new CustomResourceDefinition(s).

1. Edit deployment with `kubectl edit deploy rook-edgefs-operator -n rook-edgefs-system` and set to a new version
2. Apply new CRDs or changes. Look at examples in cluster/examples/kubernetes/edgefs/operator.yaml.
3. Monitor operator logs with `kubectl logs rook-edgefs-operator-ID -n rook-edgefs-system`

Cassandra Cluster CRD

Cassandra database clusters can be created and configuring using the `clusters.cassandra.rook.io` custom resource definition (CRD).

Please refer to the the user guide walk-through for complete instructions. This page will explain all the available configuration options on the Cassandra CRD.

Sample

```yaml
apiVersion: cassandra.rook.io/v1alpha1
kind: Cluster
metadata:
  name: rook-cassandra
  namespace: rook-cassandra
spec:
  version: 3.11.6
  repository: my-private-repo.io/cassandra
  mode: cassandra
  # A key/value list of annotations
  annotations:
  #  key: value
  datacenter:
    name: us-east-1
    racks:
      - name: us-east-1a
        members: 3
        storage:
          volumeClaimTemplates:
            - metadata:
                name: rook-cassandra-data
              spec:
                storageClassName: my-storage-class
                resources:
                  requests:
                    storage: 200Gi
        resources:
          requests:
            cpu: 8
            memory: 32Gi
          limits:
            cpu: 8
            memory: 32Gi
        # A key/value list of annotations
        annotations:
        #  key: value
        placement:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
```

```
            nodeSelectorTerms:
              - matchExpressions:
                - key: failure-domain.beta.kubernetes.io/region
                  operator: In
                  values:
                    - us-east-1
                - key: failure-domain.beta.kubernetes.io/zone
                  operator: In
                  values:
                    - us-east-1a
```

Settings Explanation

Cluster Settings

- `version`: The version of Cassandra to use. It is used as the image tag to pull.
- `repository`: Optional field. Specifies a custom image repo. If left unset, the official docker hub repo is used.
- `mode`: Optional field. Specifies if this is a Cassandra or Scylla cluster. If left unset, it defaults to cassandra. Values: {scylla, cassandra}
- `annotations`: Key value pair list of annotations to add.

In the Cassandra model, each cluster contains datacenters and each datacenter contains racks. At the moment, the operator only supports single datacenter setups.

Datacenter Settings

- `name`: Name of the datacenter. Usually, a datacenter corresponds to a region.
- `racks`: List of racks for the specific datacenter.

Rack Settings

- `name`: Name of the rack. Usually, a rack corresponds to an availability zone.
- `members`: Number of Cassandra members for the specific rack. (In Cassandra documentation, they are called nodes. We don't call them nodes to avoid confusion as a Cassandra Node corresponds to a Kubernetes Pod, not a Kubernetes Node).
- `storage`: Defines the volumes to use for each Cassandra member. Currently, only 1 volume is supported.
- `jmxExporterConfigMapName`: Name of configmap that will be used for jmx_exporter. Exporter listens on port 9180. If the name not specified, the exporter will not be run.
- `resources`: Defines the CPU and RAM resources for the Cassandra Pods.
- `annotations`: Key value pair list of annotations to add.
- `placement`: Defines the placement of Cassandra Pods. Has the following subfields:
  - `nodeAffinity`
  - `podAffinity`

- `podAntiAffinity`
- `tolerations`

Cassandra Operator Upgrades

This guide will walk you through the manual steps to upgrade the software in Cassandra Operator from one version to the next. The cassandra operator is made up of two parts:

1. The `Operator` binary that runs as a standalone application, watches the Cassandra Cluster CRD and makes administrative decisions.
2. A sidecar that runs alongside each member of a Cassandra Cluster. We will call this component `Sidecar`.

Both components should be updated. This is a very manual process at the moment, but it should be automated soon in the future, once the Cassandra Operator reaches the beta stage.

Considerations

With this upgrade guide, there are a few notes to consider:

- **WARNING**: Upgrading a Rook cluster is not without risk. There may be unexpected issues or obstacles that damage the integrity and health of your storage cluster, including data loss. Only proceed with this guide if you are comfortable with that. It is recommended that you backup your data before proceeding.
- **WARNING**: The current process to upgrade REQUIRES the cluster to be unavailable for the time of the upgrade.

Prerequisites

- If you are upgrading from v0.9.2 to a later version, the mount point of the PVC for each member has changed because it was wrong. Please follow the migration instructions for upgrading from v0.9.2.
- Before starting the procedure, ensure that your Cassandra Clusters are in a healthy state. You can check a Cassandra Cluster's health by using `kubectl describe clusters.cassandra.rook.io $NAME -n $NAMESPACE` and ensuring that for each rack in the Status, `readyMembers` equals `members`.

Procedure

1. Because each version of the `Operator` is designed to work with the same version of the `Sidecar`, they must be upgraded together. In order to avoid mixing versions between the `Operator` and `Sidecar`, we first delete every Cassandra Cluster CRD in our Kubernetes cluster, after first backing up their manifests. This will not delete your data because the PVCs will be retained even if the Cassandra Cluster object is deleted. Example:

```
# Assumes cluster rook-cassandra in namespace rook-cassandra
NAME=rook-cassandra
NAMESPACE=rook-cassandra

kubectl get clusters.cassandra.rook.io $NAME -n $NAMESPACE -o yaml > $NAME.yaml
kubectl delete clusters.cassandra.rook.io $NAME -n $NAMESPACE
```

1. After that, we upgrade the version of the `Operator`. To achieve that, we patch the StatefulSet running the `Operator`:

```
# Assumes Operator is running in StatefulSet rook-cassandra-operator
# in namespace rook-cassandra-system

kubectl set image sts/rook-cassandra-operator rook-cassandra-operator=rook/cassandra:v0.9.x -n rook-cassandra-
system
```

After patching, ensure that the operator pods are running successfully:

```
kubectl get pods -n rook-cassandra-system
```

1. Recreate the manifests previously deleted:

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

The `Operator` will pick up the newly created Cassandra Clusters and recreate them with the correct version of the sidecar.

Before Upgrading from v0.9.2

Do the following before proceeding:

- For each member of each cluster:

```
POD=rook-cassandra-us-east-1-us-east-1a-0
NAMESPACE=rook-cassandra

# Change /var/lib/cassandra to /var/lib/scylla for a scylla cluster
kubectl exec $POD -n $NAMESPACE -- /bin/bash

> mkdir /var/lib/cassandra/data/data
> shopt -s extglob
> mv !(/var/lib/cassandra/data) /var/lib/cassandra/data/data
```

After that continue with the upgrade procedure.

CockroachDB Cluster CRD

CockroachDB database clusters can be created and configured using the `clusters.cockroachdb.rook.io` custom resource definition (CRD). Please refer to the the user guide walk-through for complete instructions. This page will explain all the available configuration options on the CockroachDB CRD.

Sample

```yaml
apiVersion: cockroachdb.rook.io/v1alpha1
kind: Cluster
metadata:
  name: rook-cockroachdb
  namespace: rook-cockroachdb
spec:
  scope:
    nodeCount: 3
    volumeClaimTemplates:
      - spec:
          accessModes: [ "ReadWriteOnce" ]
          # Uncomment and specify your StorageClass, otherwise
          # the cluster admin defined default StorageClass will be used.
          #storageClassName: "my-storage-class"
          resources:
            requests:
              storage: "100Gi"
  network:
    ports:
    - name: http
      port: 8080
    - name: grpc
      port: 26257
  secure: false
  cachePercent: 25
  maxSQLMemoryPercent: 25
  # A key/value list of annotations
  annotations:
  #  key: value
```

Cluster Settings

CockroachDB Specific Settings

The settings below are specific to CockroachDB database clusters:

- `secure`: `true` to create a secure cluster installation using certificates and

encryption. `false` to create an insecure installation (strongly discouraged for production usage). Currently, only insecure is supported.
- `cachePercent`: The total size used for caches, expressed as a percentage of total physical memory.
- `maxSQLMemoryPercent`: The maximum memory capacity available to store temporary data for SQL clients, expressed as a percentage of total physical memory.
- `annotations`: Key value pair list of annotations to add.

Storage Scope

Under the `scope` field, a `StorageScopeSpec` can be specified to influence the scope or boundaries of storage that the cluster will use for its underlying storage. These properties are currently supported:

- `nodeCount`: The number of CockroachDB instances to create. Some of these instances may be scheduled on the same nodes, but exactly this many instances will be created and included in the cluster.
- `volumeClaimTemplates`: A list of PersistentVolumeClaim templates which must contain only **one or no** PersistentVolumeClaim. If no PersistentVolumeClaim is given an `emptyDir` will be given, meaning the instance data will be lost when a Pod is restarted. For an example of how PersistentVolumeClaim template should look, please look at the above sample.

Network

Under the `network` field, a `NetworkSpec` can be specified that describes network related settings of the cluster. The properties that are currently supported are:

- `ports`: The port numbers to expose the CockroachDB services on, as shown in the sample above. The supported port names are:
  - `http`: The port to bind to for HTTP requests such as the UI as well as health and debug endpoints.
  - `grpc`: The main port, served by gRPC, serves Postgres-flavor SQL, internode traffic and the command line interface.

NFS Server CRD

NFS Server can be created and configured using the `nfsservers.nfs.rook.io` custom resource definition (CRD). Please refer to the user guide walk-through for complete instructions. This page will explain all the available configuration options on the NFS CRD.

Sample

The parameters to configure the NFS CRD are demonstrated in the example below which is followed by a table that explains the parameters in more detail.

Below is a very simple example that shows sharing a volume (which could be hostPath, cephFS, cephRBD, googlePD, EBS, etc.) using NFS, without any client or per export based configuration.

For a `PersistentVolumeClaim` named `googlePD-claim`, which has Read/Write permissions and no squashing, the NFS CRD instance would look like the following:

```
apiVersion: nfs.rook.io/v1alpha1
kind: NFSServer
metadata:
  name: nfs-vol
  namespace: rook
spec:
  replicas: 1
  exports:
  - name: nfs-share
    server:
      accessMode: ReadWrite
      squash: none
    persistentVolumeClaim:
      claimName: googlePD-claim
  # A key/value list of annotations
  annotations:
  #  key: value
```

Settings

The table below explains in detail each configuration option that is available in the NFS CRD.

Parameter	Description	Default
`replicas`	The number of NFS daemon to start	`1`
`annotations`	Key value pair list of annotations	`[]`

`annotations`	to add.	`[]`
`exports`	Parameters for creating an export	`<empty>`
`exports.name`	Name of the volume being shared	`<empty>`
`exports.server`	NFS server configuration	`<empty>`
`exports.server.accessMode`	Volume access modes (Reading and Writing) for the share (Valid options are `ReadOnly` , `ReadWrite` and `none`)	`ReadWrite`
`exports.server.squash`	This prevents root users connected remotely from having root privileges (valid options are `none` , `rootId` , `root` and `all`)	`none`
`exports.server.allowedClients`	Access configuration for clients that can consume the NFS volume	`<empty>`
`exports.server.allowedClients.name`	Name of the host/hosts	`<empty>`
`exports.server.allowedClients.clients`	The host or network to which the export is being shared. Valid entries for this field are host names, IP addresses, netgroups, and CIDR network addresses.	`<empty>`
`exports.server.allowedClients.accessMode`	Reading and Writing permissions for the client *(valid options are same as `exports.server.accessMode`)*	`ReadWrite`
`exports.server.allowedClients.squash`	Squash option for the client (valid options are same as `exports.server.squash`)	`none`
`exports.persistentVolumeClaim`	The PVC that will serve as the backing volume to be exported by the NFS server. Any PVC is allowed, such as host paths, CephFS, Ceph RBD, Google PD, Amazon EBS, etc..	`<empty>`
`exports.persistentVolumeClaim.claimName`	Name of the PVC	`<empty>`

*note: if `exports.server.allowedClients.accessMode` and `exports.server.allowedClients.squash` options are specified, `exports.server.accessMode` and `exports.server.squash` are overridden respectively.

Description for `volumes.allowedClients.squash` valid options are:

Option	Description
`none`	No user id squashing is performed
`rootId`	UID `0` and GID `0` are squashed to the anonymous uid and anonymous GID.
`root`	UID `0` and GID of any value are squashed to the anonymous uid and anonymous GID.
`all`	All users are squashed

The volume that needs to be exported by NFS must be attached to NFS server pod via

GCE Persistent Disk, CephFS, RBD etc. The limitations of these volumes also apply while they are shared by NFS. The limitation and other details about these volumes can be found here.

Examples

This section contains some examples for more advanced scenarios and configuration options.

Single volume exported for access by multiple clients

This example shows how to share a volume with different options for different clients accessing the share. The EBS volume (represented by a PVC) will be exported by the NFS server for client access as `/nfs-share` (note that this PVC must already exist).

The following client groups are allowed to access this share:

- `group1` with IP address `172.17.0.5` will be given Read Only access with the root user squashed.
- `group2` includes both the network range of `172.17.0.5/16` and a host named `serverX`. They will all be granted Read/Write permissions with no user squash.

```
apiVersion: nfs.rook.io/v1alpha1
kind: NFSServer
metadata:
  name: nfs-vol
  namespace: rook
spec:
  replicas: 1
  exports:
  - name: nfs-share
    server:
      allowedClients:
      - name: group1
        clients: 172.17.0.5
        accessMode: ReadOnly
        squash: root
      - name: group2
        clients:
        - 172.17.0.0/16
        - serverX
        accessMode: ReadWrite
        squash: none
    persistentVolumeClaim:
      claimName: ebs-claim
```

Multiple volumes

Multiple volumes

This section provides an example of how to share multiple volumes from one NFS server. These volumes can all be different types (e.g., Google PD and Ceph RBD). Below we will share an Amazon EBS volume as well as a CephFS volume, using differing configuration for the two:

- The EBS volume is named `share1` and is available for all clients with Read Only access and no squash.
- The CephFS volume is named `share2` and is available for all clients with Read/Write access and no squash.

```yaml
apiVersion: nfs.rook.io/v1alpha1
kind: NFSServer
metadata:
  name: nfs-multi-vol
  namespace: rook
spec:
  replicas: 1
  exports:
  - name: share1
    server:
      allowedClients:
      - name: ebs-host
        clients: all
        accessMode: ReadOnly
        squash: none
    persistentVolumeClaim:
      claimName: ebs-claim
  - name: share2
    server:
      allowedClients:
      - name: ceph-host
        clients: all
        accessMode: ReadWrite
        squash: none
    persistentVolumeClaim:
      claimName: cephfs-claim
```

YugabyteDB Cluster CRD

YugabyteDB clusters can be created/configured by creating/updating the custom resource object `ybclusters.yugabytedb.rook.io`. Please follow instructions in the YugabyteDB Operator Quikstart to create a YugabyteDB cluster.

The configuration options provided by the custom resource are explained here.

Sample

```
apiVersion: yugabytedb.rook.io/v1alpha1
kind: YBCluster
metadata:
  name: rook-yugabytedb
  namespace: rook-yugabytedb
spec:
  master:
    # Replica count for Master.
    replicas: 3
    # optional. Default values for resource are as below
    resource:
      requests:
        cpu: 2
        memory: 2Gi
      limits:
        cpu: 2
        memory: 2Gi
    # Mentioning network ports is optional. If some or all ports are not specified, then they will be defaulted to below-mentioned values, except for tserver-ui.
    network:
      ports:
        - name: yb-master-ui
          port: 7000          # default value
        - name: yb-master-rpc
          port: 7100          # default value
    # Volume claim template for Master
    volumeClaimTemplate:
      metadata:
        name: datadir
      spec:
        accessModes: [ "ReadWriteOnce" ]
        resources:
          requests:
            storage: 1Gi
        storageClassName: standard          # Modify this field with required storage class name.
  tserver:
    # Replica count for TServer
    replicas: 3
    # optional. Default values for resource are as below
```

```
    resource:
      requests:
        cpu: 2
        memory: 4Gi
      limits:
        cpu: 2
        memory: 4Gi
    # Mentioning network ports is optional. If some or all ports are not specified, then they will be defaulted to below-mentioned values, except for tserver-ui.
    # For tserver-ui a cluster ip service will be created if the yb-tserver-ui port is explicitly mentioned. If it is not specified, only StatefulSet & headless service will be created for TServer. TServer ClusterIP service creation will be skipped. Whereas for Master, all 3 kubernetes objects will always be created.
    network:
      ports:
        - name: yb-tserver-ui
          port: 9000
        - name: yb-tserver-rpc
          port: 9100          # default value
        - name: ycql
          port: 9042          # default value
        - name: yedis
          port: 6379          # default value
        - name: ysql
          port: 5433          # default value
    # Volume claim template for TServer
    volumeClaimTemplate:
      metadata:
        name: datadir
      spec:
        accessModes: [ "ReadWriteOnce" ]
        resources:
          requests:
            storage: 1Gi
        storageClassName: standard          # Modify this field with required storage class name.
```

Configuration options

Master/TServer

Master & TServer are two essential components of a YugabyteDB cluster. Master is responsible for recording and maintaining system metadata & for admin activities. TServers are mainly responsible for data I/O. Specify Master/TServer specific attributes under `master` / `tserver` . The valid attributes are `replicas` , `network` & `volumeClaimTemplate` .

Replica Count

Specify replica count for `master` & `tserver` pods under `replicas` field. This is a **required** field.

Resource

Specify resource requests and limits for CPU & Memory. If provided, the given resource values will be used. If omitted the default CPU request & limit will be 2, whereas default memory request & limit will be 2Gi & 4Gi for Master & TServer, respectively. You may override these values for dev environments where the cluster doesn't have CPU/Memory as per the defaults. **Though it is recommended to use the defaults for production-like deployments.**

Network

`network` field accepts `NetworkSpec` to be specified which describes YugabyteDB network settings. This is an **optional** field. Default network settings will be used, if any or all of the acceptable values are absent.

A ClusterIP service will be created when `yb-tserver-ui` port is explicitly specified. If it is not specified, only StatefulSet & headless service will be created for TServer. ClusterIP service creation will be skipped. Whereas for Master, all 3 kubernetes objects will always be created.

The acceptable port names & their default values are as follows:

Name	Default Value
`yb-master-ui`	`7000`
`yb-master-rpc`	`7100`
`yb-tserver-rpc`	`9100`
`ycql`	`9042`
`yedis`	`6379`
`ysql`	`5433`

Volume Claim Templates

Specify a `PersistentVolumeClaim` template under the `volumeClaimTemplate` field for `master` & `tserver` each. This is a **required** field.

Helm Charts

Rook has published a Helm chart for the operator. Other Helm charts will also be potentially developed for each of the CRDs for all Rook storage backends.

- Rook Ceph Operator: Installs the Rook Operator and Agents necessary to run a Ceph cluster

Contributions are welcome to create our other Helm charts!

Ceph Operator Helm Chart

Installs rook to create, configure, and manage Ceph clusters on Kubernetes.

Introduction

This chart bootstraps a rook-ceph-operator deployment on a Kubernetes cluster using the Helm package manager.

Prerequisites

- Kubernetes 1.11+

RBAC

If role-based access control (RBAC) is enabled in your cluster, you may need to give Tiller (the server-side component of Helm) additional permissions. **If RBAC is not enabled, be sure to set `rbacEnable` to `false` when installing the chart.**

```
# Create a ServiceAccount for Tiller in the `kube-system` namespace
kubectl --namespace kube-system create sa tiller

# Create a ClusterRoleBinding for Tiller
kubectl create clusterrolebinding tiller --clusterrole cluster-admin --serviceaccount=kube-system:tiller

# Patch Tiller's Deployment to use the new ServiceAccount
kubectl --namespace kube-system patch deploy/tiller-deploy -p '{"spec": {"template": {"spec":
{"serviceAccountName": "tiller"}}}}'
```

Installing

The Ceph Operator helm chart will install the basic components necessary to create a storage platform for your Kubernetes cluster. After the helm chart is installed, you will need to create a Rook cluster.

The `helm install` command deploys rook on the Kubernetes cluster in the default configuration. The configuration section lists the parameters that can be configured during installation. It is recommended that the rook operator be installed into the `rook-ceph` namespace (you will install your clusters into separate namespaces).

Rook currently publishes builds of the Ceph operator to the `release` and `master` channels.

Release

The release channel is the most recent release of Rook that is considered stable for the community.

```
1. helm repo add rook-release https://charts.rook.io/release
2. helm install --namespace rook-ceph rook-release/rook-ceph
```

Master

The master channel includes the latest commits, with all automated tests green. Historically it has been very stable, though it is only recommended for testing. The critical point to consider is that upgrades are not supported to or from master builds.

To install the helm chart from master, you will need to pass the specific version returned by the `search` command.

```
1. helm repo add rook-master https://charts.rook.io/master
2. helm search rook-ceph
3. helm install --namespace rook-ceph rook-master/rook-ceph --version <version>
```

For example:

```
1. helm install --namespace rook-ceph rook-master/rook-ceph --version v0.7.0-278.gcbd9726
```

Development Build

To deploy from a local build from your development environment:

1. Build the Rook docker image: `make`
2. Copy the image to your K8s cluster, such as with the `docker save` then the `docker load` commands
3. Install the helm chart:

```
1. cd cluster/charts/rook-ceph
2. helm install --namespace rook-ceph --name rook-ceph .
```

Uninstalling the Chart

To uninstall/delete the `rook-ceph` deployment:

```
1. helm delete --purge rook-ceph
```

The command removes all the Kubernetes components associated with the chart and deletes the release.

Configuration

The following tables lists the configurable parameters of the rook-operator chart and their default values.

Parameter	Description	Default
`image.repository`	Image	`rook/ceph`
`image.tag`	Image tag	`master`
`image.pullPolicy`	Image pull policy	`IfNotPresent`
`rbacEnable`	If true, create & use RBAC resources	`true`
`pspEnable`	If true, create & use PSP resources	`true`
`resources`	Pod resource requests & limits	`{}`
`annotations`	Pod annotations	`{}`
`logLevel`	Global log level	`INFO`
`nodeSelector`	Kubernetes `nodeSelector` to add to the Deployment.	
`tolerations`	List of Kubernetes `tolerations` to add to the Deployment.	`[]`
`unreachableNodeTolerationSeconds`	Delay to use for the node.kubernetes.io/unreachable pod failure toleration to override the Kubernetes default of 5 minutes	`5s`
`currentNamespaceOnly`	Whether the operator should watch cluster CRD in its own namespace or not	`false`
`hostpathRequiresPrivileged`	Runs Ceph Pods as privileged to be able to write to `hostPath` s in OpenShift with SELinux restrictions.	`false`
`mon.healthCheckInterval`	The frequency for the operator to check the mon health	`45s`
`mon.monOutTimeout`	The time to wait before failing over an unhealthy mon	`600s`
`discover.priorityClassName`	The priority class name to add to the discover pods	
`discover.toleration`	Toleration for the discover pods	
`discover.tolerationKey`	The specific key of the taint to tolerate	
`discover.tolerations`	Array of tolerations in YAML format which will be added to discover deployment	
`discover.nodeAffinity`	The node labels for affinity of `discover-agent` ()	

`csi.enableRbdDriver`	Enable Ceph CSI RBD driver.	`true`
`csi.enableCephfsDriver`	Enable Ceph CSI CephFS driver.	`true`
`csi.pluginPriorityClassName`	PriorityClassName to be set on csi driver plugin pods.	
`csi.provisionerPriorityClassName`	PriorityClassName to be set on csi driver provisioner pods.	
`csi.logLevel`	Set logging level for csi containers. Supported values from 0 to 5. 0 for general useful logs, 5 for trace level verbosity.	`0`
`csi.enableGrpcMetrics`	Enable Ceph CSI GRPC Metrics.	`true`
`csi.provisionerTolerations`	Array of tolerations in YAML format which will be added to CSI provisioner deployment.	
`csi.provisionerNodeAffinity`	The node labels for affinity of the CSI provisioner deployment ()	
`csi.pluginTolerations`	Array of tolerations in YAML format which will be added to Ceph CSI plugin DaemonSet	
`csi.pluginNodeAffinity`	The node labels for affinity of the Ceph CSI plugin DaemonSet ()	
`csi.csiRBDProvisionerResource`	CEPH CSI RBD provisioner resource requirement list.	
`csi.csiRBDPluginResource`	CEPH CSI RBD plugin resource requirement list.	
`csi.csiCephFSProvisionerResource`	CEPH CSI CephFS provisioner resource requirement list.	
`csi.csiCephFSPluginResource`	CEPH CSI CephFS plugin resource requirement list.	
`csi.cephfsGrpcMetricsPort`	CSI CephFS driver GRPC metrics port.	`9091`
`csi.cephfsLivenessMetricsPort`	CSI CephFS driver metrics port.	`9081`
`csi.rbdGrpcMetricsPort`	Ceph CSI RBD driver GRPC metrics port.	`9090`
`csi.rbdLivenessMetricsPort`	Ceph CSI RBD driver metrics port.	`8080`
`csi.forceCephFSKernelClient`	Enable Ceph Kernel clients on kernel < 4.17 which support quotas for Cephfs.	`true`
`csi.kubeletDirPath`	Kubelet root directory path (if the Kubelet uses a different path for the `--root-dir` flag)	`/var/lib/kubelet`
`csi.cephcsi.image`	Ceph CSI image.	`quay.io/cephcsi/cephcsi:v3`
	CSI Rbd plugin daemonset	

`csi.rbdPluginUpdateStrategy`	update strategy, supported values are OnDelete and RollingUpdate.	`OnDelete`
`csi.cephFSPluginUpdateStrategy`	CSI CephFS plugin daemonset update strategy, supported values are OnDelete and RollingUpdate.	`OnDelete`
`csi.registrar.image`	Kubernetes CSI registrar image.	`quay.io/k8scsi/csi-node-dr registrar:v1.2.0`
`csi.resizer.image`	Kubernetes CSI resizer image.	`quay.io/k8scsi/csi- resizer:v0.4.0`
`csi.provisioner.image`	Kubernetes CSI provisioner image.	`quay.io/k8scsi/csi- provisioner:v1.6.0`
`csi.snapshotter.image`	Kubernetes CSI snapshotter image.	`quay.io/k8scsi/csi- snapshotter:v2.1.1`
`csi.attacher.image`	Kubernetes CSI Attacher image.	`quay.io/k8scsi/csi- attacher:v2.1.0`
`agent.flexVolumeDirPath`	Path where the Rook agent discovers the flex volume plugins ()	`/usr/libexec/kubernetes/ku plugins/volume/exec/`
`agent.libModulesDirPath`	Path where the Rook agent should look for kernel modules (*)	`/lib/modules`
`agent.mounts`	Additional paths to be mounted in the agent container ()	
`agent.mountSecurityMode`	Mount Security Mode for the agent.	`Any`
`agent.priorityClassName`	The priority class name to add to the agent pods	
`agent.toleration`	Toleration for the agent pods	
`agent.tolerationKey`	The specific key of the taint to tolerate	
`agent.tolerations`	Array of tolerations in YAML format which will be added to agent deployment	
`agent.nodeAffinity`	The node labels for affinity of `rook-agent` (*)	

* For information on what to set `agent.flexVolumeDirPath` to, please refer to the Rook flexvolume documentation

* * `agent.mounts` should have this format `mountname1=/host/path:/container/path,mountname2=/host/path2:/container/path2`

* * * `nodeAffinity` and `*NodeAffinity` options should have the format `"role=storage,rook; storage=ceph"` or `storage=;role=rook-example` or `storage=;` (*checks only for presence of key*)

Command Line

You can pass the settings with helm command line parameters. Specify each parameter

using the `--set key=value[,key=value]` argument to `helm install`. For example, the following command will install rook where RBAC is not enabled.

```
helm install --namespace rook-ceph --name rook-ceph rook-release/rook-ceph --set rbacEnable=false
```

Settings File

Alternatively, a yaml file that specifies the values for the above parameters (`values.yaml`) can be provided while installing the chart.

```
helm install --namespace rook-ceph --name rook-ceph rook-release/rook-ceph -f values.yaml
```

Here are the sample settings to get you started.

```
image:
  prefix: rook
  repository: rook/ceph
  tag: master
  pullPolicy: IfNotPresent

resources:
  limits:
    cpu: 100m
    memory: 256Mi
  requests:
    cpu: 100m
    memory: 256Mi

rbacEnable: true
pspEnable: true
```

Common Issues

To help troubleshoot your Rook clusters, here are some tips on what information will help solve the issues you might be seeing. If after trying the suggestions found on this page and the problem is not resolved, the Rook team is very happy to help you troubleshoot the issues in their Slack channel. Once you have registered for the Rook Slack, proceed to the General channel to ask for assistance.

Ceph

For common issues specific to Ceph, see the Ceph Common Issues page.

Troubleshooting Techniques

Kubernetes status and logs are the the main resources needed to investigate issues in any Rook cluster.

Kubernetes Tools

Kubernetes status is the first line of investigating when something goes wrong with the cluster. Here are a few artifacts that are helpful to gather:

- Rook pod status:
  - `kubectl get pod -n <cluster-namespace> -o wide`
    - e.g., `kubectl get pod -n rook-ceph -o wide`
- Logs for Rook pods
  - Logs for the operator: `kubectl logs -n <cluster-namespace> -l app=<storage-backend-operator>`
    - e.g., `kubectl logs -n rook-ceph -l app=rook-ceph-operator`
  - Logs for a specific pod: `kubectl logs -n <cluster-namespace> <pod-name>`, or a pod using a label such as mon1: `kubectl logs -n <cluster-namespace> -l <label-matcher>`
    - e.g., `kubectl logs -n rook-ceph -l mon=a`
  - Logs on a specific node to find why a PVC is failing to mount:
    - Connect to the node, then get kubelet logs (if your distro is using systemd): `journalctl -u kubelet`
  - Pods with multiple containers
    - For all containers, in order: `kubectl -n <cluster-namespace> logs <pod-name> --all-containers`
    - For a single container: `kubectl -n <cluster-namespace> logs <pod-name> -c <container-name>`
  - Logs for pods which are no longer running: `kubectl -n <cluster-namespace> logs --previous <pod-name>`

Some pods have specialized init containers, so you may need to look at logs for

different containers within the pod.

- `kubectl -n <namespace> logs <pod-name> -c <container-name>`
- Other Rook artifacts: `kubectl -n <cluster-namespace> get all`

Ceph Common Issues

Many of these problem cases are hard to summarize down to a short phrase that adequately describes the problem. Each problem will start with a bulleted list of symptoms. Keep in mind that all symptoms may not apply depending on the configuration of Rook. If the majority of the symptoms are seen there is a fair chance you are experiencing that problem.

If after trying the suggestions found on this page and the problem is not resolved, the Rook team is very happy to help you troubleshoot the issues in their Slack channel. Once you have registered for the Rook Slack, proceed to the `#ceph` channel to ask for assistance.

Table of Contents

Troubleshooting Techniques

There are two main categories of information you will need to investigate issues in the cluster:

1. Kubernetes status and logs documented here
2. Ceph cluster status (see upcoming Ceph tools section)

Ceph Tools

After you verify the basic health of the running pods, next you will want to run Ceph tools for status of the storage components. There are two ways to run the Ceph tools,

either in the Rook toolbox or inside other Rook pods that are already running.

- Logs on a specific node to find why a PVC is failing to mount:
  - Rook agent errors around the attach/detach: `kubectl logs -n rook-ceph <rook-ceph-agent-pod>`
- See the log collection topic for a script that will help you gather the logs
- Other artifacts:
  - The monitors that are expected to be in quorum: `kubectl -n <cluster-namespace> get configmap rook-ceph-mon-endpoints -o yaml | grep data`

Tools in the Rook Toolbox

The rook-ceph-tools pod provides a simple environment to run Ceph tools. Once the pod is up and running, connect to the pod to execute Ceph commands to evaluate that current state of the cluster.

```
    kubectl -n rook-ceph exec -it $(kubectl -n rook-ceph get pod -l "app=rook-ceph-tools" -o
1. jsonpath='{.items[0].metadata.name}') bash
```

Ceph Commands

Here are some common commands to troubleshoot a Ceph cluster:

- `ceph status`
- `ceph osd status`
- `ceph osd df`
- `ceph osd utilization`
- `ceph osd pool stats`
- `ceph osd tree`
- `ceph pg stat`

The first two status commands provide the overall cluster health. The normal state for cluster operations is HEALTH_OK, but will still function when the state is in a HEALTH_WARN state. If you are in a WARN state, then the cluster is in a condition that it may enter the HEALTH_ERROR state at which point *all* disk I/O operations are halted. If a HEALTH_WARN state is observed, then one should take action to prevent the cluster from halting when it enters the HEALTH_ERROR state.

There are many Ceph sub-commands to look at and manipulate Ceph objects, well beyond the scope this document. See the Ceph documentation for more details of gathering information about the health of the cluster. In addition, there are other helpful hints and some best practices located in the Advanced Configuration section. Of particular note, there are scripts for collecting logs and gathering OSD information there.

Pod Using Ceph Storage Is Not Running

> This topic is specific to creating PVCs based on Rook's **Flex** driver, which is no longer the default option. By default, Rook deploys the CSI driver for binding the PVCs to the storage.

Symptoms

- The pod that is configured to use Rook storage is stuck in the `ContainerCreating` status
- `kubectl describe pod` for the pod mentions one or more of the following:
  - `PersistentVolumeClaim is not bound`
  - `timeout expired waiting for volumes to attach/mount`
- `kubectl -n rook-ceph get pod` shows the rook-ceph-agent pods in a `CrashLoopBackOff` status

If you see that the PVC remains in **pending** state, see the topic PVCs stay in pending state.

Possible Solutions Summary

- `rook-ceph-agent` pod is in a `CrashLoopBackOff` status because it cannot deploy its driver on a read-only filesystem: Flexvolume configuration pre-reqs
- Persistent Volume and/or Claim are failing to be created and bound: Volume Creation
- `rook-ceph-agent` pod is failing to mount and format the volume: Rook Agent Mounting

Investigation Details

If you see some of the symptoms above, it's because the requested Rook storage for your pod is not being created and mounted successfully. In this walkthrough, we will be looking at the wordpress mysql example pod that is failing to start.

To first confirm there is an issue, you can run commands similar to the following and you should see similar output (note that some of it has been omitted for brevity):

```
> kubectl get pod
NAME                              READY     STATUS              RESTARTS   AGE
wordpress-mysql-918363043-50pjr   0/1       ContainerCreating   0          1h

> kubectl describe pod wordpress-mysql-918363043-50pjr
...
Events:
  FirstSeen    LastSeen    Count    From            SubObjectPath    Type        Reason            Message
  ---------    --------    -----    ----            -------------    --------    ------            -------
  1h        1h        3    default-scheduler            Warning        FailedScheduling
PersistentVolumeClaim is not bound: "mysql-pv-claim" (repeated 2 times)
  1h        35s        36    kubelet, 172.17.8.101            Warning        FailedMount        Unable to
mount volumes for pod "wordpress-mysql-918363043-50pjr_default(08d14e75-bd99-11e7-bc4c-001c428b9fc8)": timeout
expired waiting for volumes to attach/mount for pod "default"/"wordpress-mysql-918363043-50pjr". list of
unattached/unmounted volumes=[mysql-persistent-storage]
  1h        35s        36    kubelet, 172.17.8.101            Warning        FailedSync        Error syncing
pod
```

To troubleshoot this, let’s walk through the volume provisioning steps in order to confirm where the failure is happening.

Ceph Agent Deployment

The `rook-ceph-agent` pods are responsible for mapping and mounting the volume from the cluster onto the node that your pod will be running on. If the `rook-ceph-agent` pod is not running then it cannot perform this function.

Below is an example of the `rook-ceph-agent` pods failing to get to the `Running` status because they are in a `CrashLoopBackOff` status:

```
> kubectl -n rook-ceph get pod
NAME                                  READY     STATUS             RESTARTS   AGE
rook-ceph-agent-ct5pj                 0/1       CrashLoopBackOff   16         59m
rook-ceph-agent-zb6n9                 0/1       CrashLoopBackOff   16         59m
rook-operator-2203999069-pmhzn        1/1       Running            0          59m
```

If you see this occurring, you can get more details about why the `rook-ceph-agent` pods are continuing to crash with the following command and its sample output:

```
> kubectl -n rook-ceph get pod -l app=rook-ceph-agent -o jsonpath='{range .items[*]}{.metadata.name}{"\t"}
{.status.containerStatuses[0].lastState.terminated.message}{"\n"}{end}'
rook-ceph-agent-ct5pj    mkdir /usr/libexec/kubernetes: read-only filesystem
rook-ceph-agent-zb6n9    mkdir /usr/libexec/kubernetes: read-only filesystem
```

From the output above, we can see that the agents were not able to bind mount to `/usr/libexec/kubernetes` on the host they are scheduled to run on. For some environments, this default path is read-only and therefore a better path must be provided to the agents.

First, clean up the agent deployment with:

```
kubectl -n rook-ceph delete daemonset rook-ceph-agent
```

Once the `rook-ceph-agent` pods are gone, **follow the instructions in the Flexvolume configuration pre-reqs** to ensure a good value for `--volume-plugin-dir` has been provided to the Kubelet. After that has been configured, and the Kubelet has been restarted, start the agent pods up again by restarting `rook-operator`:

```
kubectl -n rook-ceph delete pod -l app=rook-ceph-operator
```

Volume Creation

The volume must first be created in the Rook cluster and then bound to a volume claim before it can be mounted to a pod. Let’s confirm that with the following commands and their output:

```
> kubectl get pv
NAME                                       CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS     CLAIM
STORAGECLASS   REASON    AGE
pvc-9f273fbc-bdbf-11e7-bc4c-001c428b9fc8   20Gi       RWO           Delete          Bound      default/mysql-
pv-claim   rook-ceph-block               25m

> kubectl get pvc
NAME             STATUS    VOLUME                                     CAPACITY   ACCESSMODES   STORAGECLASS
AGE
mysql-pv-claim   Bound     pvc-9f273fbc-bdbf-11e7-bc4c-001c428b9fc8   20Gi       RWO           rook-ceph-block
25m
```

Both your volume and its claim should be in the `Bound` status. If one or neither of them is not in the `Bound` status, then look for details of the issue in the `rook-operator` logs:

```
kubectl -n rook-ceph logs `kubectl -n rook-ceph -l app=rook-ceph-operator get pods -o
jsonpath='{.items[*].metadata.name}'`
```

If the volume is failing to be created, there should be details in the `rook-operator` log output, especially those tagged with `op-provisioner`.

One common cause for the `rook-operator` failing to create the volume is when the `clusterNamespace` field of the `StorageClass` doesn't match the **namespace** of the Rook cluster, as described in #1502. In that scenario, the `rook-operator` log would show a failure similar to the following:

```
2018-03-28 18:58:32.041603 I | op-provisioner: creating volume with configuration {pool:replicapool
clusterNamespace:rook-ceph fstype:}
2018-03-28 18:58:32.041728 I | exec: Running command: rbd create replicapool/pvc-fd8aba49-32b9-11e8-978e-
08002762c796 --size 20480 --cluster=rook --conf=/var/lib/rook/rook-ceph/rook.config --
keyring=/var/lib/rook/rook-ceph/client.admin.keyring
E0328 18:58:32.060893       5 controller.go:801] Failed to provision volume for claim "default/mysql-pv-claim"
with StorageClass "rook-ceph-block": Failed to create rook block image replicapool/pvc-fd8aba49-32b9-11e8-
978e-08002762c796: failed to create image pvc-fd8aba49-32b9-11e8-978e-08002762c796 in pool replicapool of size
21474836480: Failed to complete '': exit status 1. global_init: unable to open config file from search list
/var/lib/rook/rook-ceph/rook.config
. output:
```

The solution is to ensure that the `clusterNamespace` field matches the **namespace** of the Rook cluster when creating the `StorageClass`.

Volume Mounting

The final step in preparing Rook storage for your pod is for the `rook-ceph-agent` pod to mount and format it. If all the preceding sections have been successful or inconclusive, then take a look at the `rook-ceph-agent` pod logs for further clues. You can determine which `rook-ceph-agent` is running on the same node that your pod is scheduled on by using the `-o wide` output, then you can get the logs for that `rook-ceph-agent` pod similar to the example below:

```
> kubectl -n rook-ceph get pod -o wide
NAME                                  READY     STATUS    RESTARTS   AGE       IP              NODE
rook-ceph-agent-h6scx                 1/1       Running   0          9m        172.17.8.102    172.17.8.102
rook-ceph-agent-mp7tn                 1/1       Running   0          9m        172.17.8.101    172.17.8.101
rook-operator-2203999069-3tb68        1/1       Running   0          9m        10.32.0.7       172.17.8.101

> kubectl -n rook-ceph logs rook-ceph-agent-h6scx
2017-10-30 23:07:06.984108 I | rook: starting Rook v0.5.0-241.g48ce6de.dirty with arguments
'/usr/local/bin/rook agent'
[...]
```

In the `rook-ceph-agent` pod logs, you may see a snippet similar to the following:

```
Failed to complete rbd: signal: interrupt.
```

In this case, the agent waited for the `rbd` command but it did not finish in a timely manner so the agent gave up and stopped it. This can happen for multiple reasons, but using `dmesg` will likely give you insight into the root cause. If `dmesg` shows something similar to below, then it means you have an old kernel that can't talk to the cluster:

```
libceph: mon2 10.205.92.13:6789 feature set mismatch, my 4a042a42 < server's 2004a042a42, missing 20000000000
```

If `uname -a` shows that you have a kernel version older than `3.15`, you'll need to perform **one** of the following:

- Disable some Ceph features by starting the rook toolbox and running `ceph osd crush tunables bobtail`
- Upgrade your kernel to `3.15` or later.

Filesystem Mounting

In the `rook-ceph-agent` pod logs, you may see a snippet similar to the following:

```
2017-11-07 00:04:37.808870 I | rook-flexdriver: WARNING: The node kernel version is 4.4.0-87-generic, which do
not support multiple ceph filesystems. The kernel version has to be at least 4.7. If you have multiple ceph
filesystems, the result could be inconsistent
```

This will happen in kernels with versions older than 4.7, where the option `mds_namespace` is not supported. This option is used to specify a filesystem namespace.

In this case, if there is only one filesystem in the Rook cluster, there should be no issues and the mount should succeed. If you have more than one filesystem, inconsistent results may arise and the filesystem mounted may not be the one you specified.

If the issue is still not resolved from the steps above, please come chat with us on the **#general** channel of our Rook Slack. We want to help you get your storage working

and learn from those lessons to prevent users in the future from seeing the same issue.

Cluster failing to service requests

Symptoms

- Execution of the `ceph` command hangs
- PersistentVolumes are not being created
- Large amount of slow requests are blocking
- Large amount of stuck requests are blocking
- One or more MONs are restarting periodically

Investigation

Create a rook-ceph-tools pod to investigate the current state of Ceph. Here is an example of what one might see. In this case the `ceph status` command would just hang so a CTRL-C needed to be sent.

```
$ kubectl -n rook-ceph exec -it $(kubectl -n rook-ceph get pod -l "app=rook-ceph-tools" -o
jsonpath='{.items[0].metadata.name}') bash
root@rook-ceph-tools:/# ceph status
^CCluster connection interrupted or timed out
```

Another indication is when one or more of the MON pods restart frequently. Note the 'mon107' that has only been up for 16 minutes in the following output.

```
$ kubectl -n rook-ceph get all -o wide --show-all
NAME                                  READY     STATUS    RESTARTS   AGE       IP               NODE
po/rook-ceph-mgr0-2487684371-gzlbq    1/1       Running   0          17h       192.168.224.46   k8-host-0402
po/rook-ceph-mon107-p74rj             1/1       Running   0          16m       192.168.224.28   k8-host-0402
rook-ceph-mon1-56fgm                  1/1       Running   0          2d        192.168.91.135   k8-host-0404
rook-ceph-mon2-rlxcd                  1/1       Running   0          2d        192.168.123.33   k8-host-0403
rook-ceph-osd-bg2vj                   1/1       Running   0          2d        192.168.91.177   k8-host-0404
rook-ceph-osd-mwxdm                   1/1       Running   0          2d        192.168.123.31   k8-host-0403
```

Solution

What is happening here is that the MON pods are restarting and one or more of the Ceph daemons are not getting configured with the proper cluster information. This is commonly the result of not specifying a value for `dataDirHostPath` in your Cluster CRD.

The `dataDirHostPath` setting specifies a path on the local host for the Ceph daemons to store configuration and data. Setting this to a path like `/var/lib/rook`, reapplying your Cluster CRD and restarting all the Ceph daemons (MON, MGR, OSD, RGW) should solve this problem. After the Ceph daemons have been restarted, it is advisable to restart

the rook-tool pod.

Monitors are the only pods running

Symptoms

- Rook operator is running
- Either a single mon starts or the mons skip letters, specifically named `a`, `d`, and `f`
- No mgr, osd, or other daemons are created

Investigation

When the operator is starting a cluster, the operator will start one mon at a time and check that they are healthy before continuing to bring up all three mons. If the first mon is not detected healthy, the operator will continue to check until it is healthy. If the first mon fails to start, a second and then a third mon may attempt to start. However, they will never form quorum and the orchestration will be blocked from proceeding.

The likely causes for the mon health not being detected:

- The operator pod does not have network connectivity to the mon pod
- The mon pod is failing to start
- One or more mon pods are in running state, but are not able to form quorum

Operator fails to connect to the mon

First look at the logs of the operator to confirm if it is able to connect to the mons.

```
kubectl -n rook-ceph logs -l app=rook-ceph-operator
```

Likely you will see an error similar to the following that the operator is timing out when connecting to the mon. The last command is `ceph mon_status`, followed by a timeout message five minutes later.

```
2018-01-21 21:47:32.375833 I | exec: Running command: ceph mon_status --cluster=rook --
conf=/var/lib/rook/rook-ceph/rook.config --keyring=/var/lib/rook/rook-ceph/client.admin.keyring --format json
--out-file /tmp/442263890
2018-01-21 21:52:35.370533 I | exec: 2018-01-21 21:52:35.071462 7f96a3b82700  0 monclient(hunting):
authenticate timed out after 300
2018-01-21 21:52:35.071462 7f96a3b82700  0 monclient(hunting): authenticate timed out after 300
2018-01-21 21:52:35.071524 7f96a3b82700  0 librados: client.admin authentication error (110) Connection timed
out
2018-01-21 21:52:35.071524 7f96a3b82700  0 librados: client.admin authentication error (110) Connection timed
out
[errno 110] error connecting to the cluster
```

The error would appear to be an authentication error, but it is misleading. The real issue is a timeout.

Solution

If you see the timeout in the operator log, verify if the mon pod is running (see the next section). If the mon pod is running, check the network connectivity between the operator pod and the mon pod. A common issue is that the CNI is not configured correctly.

Failing mon pod

Second we need to verify if the mon pod started successfully.

```
$ kubectl -n rook-ceph get pod -l app=rook-ceph-mon
NAME                               READY     STATUS             RESTARTS   AGE
rook-ceph-mon-a-69fb9c78cd-58szd   1/1       CrashLoopBackOff   2          47s
```

If the mon pod is failing as in this example, you will need to look at the mon pod status or logs to determine the cause. If the pod is in a crash loop backoff state, you should see the reason by describing the pod.

```
# The pod shows a termination status that the keyring does not match the existing keyring
$ kubectl -n rook-ceph describe pod -l mon=rook-ceph-mon0
...
    Last State:    Terminated
      Reason:    Error
      Message:    The keyring does not match the existing keyring in /var/lib/rook/rook-ceph-
mon0/data/keyring.
                  You may need to delete the contents of dataDirHostPath on the host from a previous
deployment.
...
```

See the solution in the next section regarding cleaning up the `dataDirHostPath` on the nodes.

Three mons named a, d, and f

If you see the three mons running with the names `a`, `d`, and `f`, they likely did not form quorum even though they are running.

```
NAME                               READY   STATUS    RESTARTS   AGE
rook-ceph-mon-a-7d9fd97d9b-cdq7g   1/1     Running   0          10m
rook-ceph-mon-d-77df8454bd-r5jwr   1/1     Running   0          9m2s
rook-ceph-mon-f-58b4f8d9c7-89lgs   1/1     Running   0          7m38s
```

Solution

This is a common problem reinitializing the Rook cluster when the local directory used for persistence has **not** been purged. This directory is the `dataDirHostPath` setting in the cluster CRD and is typically set to `/var/lib/rook`. To fix the issue you will need to delete all components of Rook and then delete the contents of `/var/lib/rook` (or the directory specified by `dataDirHostPath`) on each of the hosts in the cluster. Then when the cluster CRD is applied to start a new cluster, the rook-operator should start all the pods as expected.

> IMPORTANT: Deleting the `dataDirHostPath` folder is destructive to the storage. Only delete the folder if you are trying to permanently purge the Rook cluster.

See the Cleanup Guide for more details.

PVCs stay in pending state

Symptoms

- When you create a PVC based on a rook storage class, it stays pending indefinitely

For the Wordpress example, you might see two PVCs in pending state.

```
$ kubectl get pvc
NAME             STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS      AGE
mysql-pv-claim   Pending                                      rook-ceph-block   8s
wp-pv-claim      Pending                                      rook-ceph-block   16s
```

Investigation

There are two common causes for the PVCs staying in pending state:

1. There are no OSDs in the cluster
2. The CSI provisioner pod is not running or is not responding to the request to provision the storage

If you are still using the Rook flex driver for the volumes (the CSI driver is the default since Rook v1.1), another cause could be that the operator is not running or is otherwise not responding to the request to provision the storage.

Confirm if there are OSDs

To confirm if you have OSDs in your cluster, connect to the Rook Toolbox and run the `ceph status` command. You should see that you have at least one OSD `up` and `in`. The minimum number of OSDs required depends on the `replicated.size` setting in the pool created for the storage class. In a “test” cluster, only one OSD is required (see `storageclass-test.yaml`). In the production storage class example (`storageclass.yaml`), three OSDs would be required.

```
$ ceph status
  cluster:
    id:     a0452c76-30d9-4c1a-a948-5d8405f19a7c
    health: HEALTH_OK

  services:
    mon: 3 daemons, quorum a,b,c (age 11m)
    mgr: a(active, since 10m)
    osd: 1 osds: 1 up (since 46s), 1 in (since 109m)
```

OSD Prepare Logs

If you don't see the expected number of OSDs, let's investigate why they weren't created. On each node where Rook looks for OSDs to configure, you will see an "osd prepare" pod.

```
$ kubectl -n rook-ceph get pod -l app=rook-ceph-osd-prepare
NAME                                  ...  READY   STATUS      RESTARTS   AGE
rook-ceph-osd-prepare-minikube-9twvk   0/2     Completed   0          30m
```

See the section on why OSDs are not getting created to investigate the logs.

CSI Driver

The CSI driver may not be responding to the requests. Look in the logs of the CSI provisioner pod to see if there are any errors during the provisioning.

There are two provisioner pods:

```
kubectl -n rook-ceph get pod -l app=csi-rbdplugin-provisioner
```

Get the logs of each of the pods. One of them should be the "leader" and be responding to requests.

```
kubectl -n rook-ceph logs csi-cephfsplugin-provisioner-d77bb49c6-q9hwq csi-provisioner
```

Operator unresponsiveness

Lastly, if you have OSDs `up` and `in`, the next step is to confirm the operator is responding to the requests. Look in the Operator pod logs around the time when the PVC was created to confirm if the request is being raised. If the operator does not show requests to provision the block image, the operator may be stuck on some other operation. In this case, restart the operator pod to get things going again.

Solution

If the “osd prepare” logs didn’t give you enough clues about why the OSDs were not being created, please review your cluster.yaml configuration. The common misconfigurations include:

- If `useAllDevices: true`, Rook expects to find local devices attached to the nodes. If no devices are found, no OSDs will be created.
- If `useAllDevices: false`, OSDs will only be created if `deviceFilter` is specified.
- Only local devices attached to the nodes will be configurable by Rook. In other words, the devices must show up under `/dev`.
    - The devices must not have any partitions or filesystems on them. Rook will only configure raw devices. Partitions are not yet supported.

OSD pods are failing to start

Symptoms

- OSD pods are failing to start
- You have started a cluster after tearing down another cluster

Investigation

When an OSD starts, the device or directory will be configured for consumption. If there is an error with the configuration, the pod will crash and you will see the CrashLoopBackoff status for the pod. Look in the osd pod logs for an indication of the failure.

```
$ kubectl -n rook-ceph logs rook-ceph-osd-fl8fs
...
```

One common case for failure is that you have re-deployed a test cluster and some state may remain from a previous deployment. If your cluster is larger than a few nodes, you may get lucky enough that the monitors were able to start and form quorum. However, now the OSDs pods may fail to start due to the old state. Looking at the OSD pod logs you will see an error about the file already existing.

```
$ kubectl -n rook-ceph logs rook-ceph-osd-fl8fs
...
2017-10-31 20:13:11.187106 I | mkfs-osd0: 2017-10-31 20:13:11.186992 7f0059d62e00 -1
bluestore(/var/lib/rook/osd0) _read_fsid unparsable uuid
2017-10-31 20:13:11.187208 I | mkfs-osd0: 2017-10-31 20:13:11.187026 7f0059d62e00 -1
bluestore(/var/lib/rook/osd0) _setup_block_symlink_or_file failed to create block symlink to /dev/disk/by-
partuuid/651153ba-2dfc-4231-ba06-94759e5ba273: (17) File exists
2017-10-31 20:13:11.187233 I | mkfs-osd0: 2017-10-31 20:13:11.187038 7f0059d62e00 -1
bluestore(/var/lib/rook/osd0) mkfs failed, (17) File exists
2017-10-31 20:13:11.187254 I | mkfs-osd0: 2017-10-31 20:13:11.187042 7f0059d62e00 -1 OSD::mkfs:
ObjectStore::mkfs failed with error (17) File exists
2017-10-31 20:13:11.187275 I | mkfs-osd0: 2017-10-31 20:13:11.187121 7f0059d62e00 -1  ** ERROR: error creating
empty object store in /var/lib/rook/osd0: (17) File exists
```

Solution

If the error is from the file that already exists, this is a common problem reinitializing the Rook cluster when the local directory used for persistence has **not** been purged. This directory is the `dataDirHostPath` setting in the cluster CRD and is typically set to `/var/lib/rook`. To fix the issue you will need to delete all components of Rook and then delete the contents of `/var/lib/rook` (or the directory specified by `dataDirHostPath`) on each of the hosts in the cluster. Then when the cluster CRD is applied to start a new cluster, the rook-operator should start all the pods as expected.

OSD pods are not created on my devices

Symptoms

- No OSD pods are started in the cluster
- Devices are not configured with OSDs even though specified in the Cluster CRD
- One OSD pod is started on each node instead of multiple pods for each device

Investigation

First, ensure that you have specified the devices correctly in the CRD. The Cluster CRD has several ways to specify the devices that are to be consumed by the Rook storage:

- `useAllDevices: true`: Rook will consume all devices it determines to be available
- `deviceFilter`: Consume all devices that match this regular expression
- `devices`: Explicit list of device names on each node to consume

Second, if Rook determines that a device is not available (has existing partitions or a formatted filesystem), Rook will skip consuming the devices. If Rook is not starting OSDs on the devices you expect, Rook may have skipped it for this reason. To see if a device was skipped, view the OSD preparation log on the node where the device was skipped. Note that it is completely normal and expected for OSD prepare pod to be in the `completed` state. After the job is complete, Rook leaves the pod around in case the logs need to be investigated.

```
# Get the prepare pods in the cluster
$ kubectl -n rook-ceph get pod -l app=rook-ceph-osd-prepare
NAME                                READY     STATUS      RESTARTS   AGE
rook-ceph-osd-prepare-node1-fvmrp   0/1       Completed   0          18m
rook-ceph-osd-prepare-node2-w9xv9   0/1       Completed   0          22m
rook-ceph-osd-prepare-node3-7rgnv   0/1       Completed   0          22m

# view the logs for the node of interest in the "provision" container
$ kubectl -n rook-ceph logs rook-ceph-osd-prepare-node1-fvmrp provision
[...]
```

Here are some key lines to look for in the log:

```
# A device will be skipped if Rook sees it has partitions or a filesystem
2019-05-30 19:02:57.353171 W | cephosd: skipping device sda that is in use
2019-05-30 19:02:57.452168 W | skipping device "sdb5": ["Used by ceph-disk"]

# Other messages about a disk being unusable by ceph include:
Insufficient space (<5GB) on vgs
Insufficient space (<5GB)
LVM detected
Has BlueStore device label
locked
read-only

# A device is going to be configured
2019-05-30 19:02:57.535598 I | cephosd: device sdc to be configured by ceph-volume

# For each device configured you will see a report printed to the log
2019-05-30 19:02:59.844642 I |   Type            Path                                                    LV
Size         % of device
2019-05-30 19:02:59.844651 I | --------------------------------------------------------------------------------
----------------------
2019-05-30 19:02:59.844677 I |   [data]          /dev/sdc                                                7.00
GB         100%
```

Solution

Either update the CR with the correct settings, or clean the partitions or filesystem from your devices. To clean devices from a previous install see the cleanup guide.

After the settings are updated or the devices are cleaned, trigger the operator to analyze the devices again by restarting the operator. Each time the operator starts, it will ensure all the desired devices are configured. The operator does automatically deploy OSDs in most scenarios, but an operator restart will cover any scenarios that the operator doesn't detect automatically.

```
# Restart the operator to ensure devices are configured. A new pod will automatically be started when the
current operator pod is deleted.
$ kubectl -n rook-ceph delete pod -l app=rook-ceph-operator
[...]
```

Node hangs after reboot

This issue is fixed in `cephcsi:v2.0.1` and newer.

Symptoms

- After issuing a `reboot` command, node never returned online

- Only a power cycle helps

Investigation

On a node running a pod with a Ceph persistent volume

```
$ mount | grep rbd

# _netdev mount option is absent, also occurs for cephfs
# OS is not aware PV is mounted over network
/dev/rbdx on ... (rw,relatime, ..., noquota)
```

When the reboot command is issued, network interfaces are terminated before disks are unmounted. This results in the node hanging as repeated attempts to unmount Ceph persistent volumes fail with the following error:

```
libceph: connect [monitor-ip]:6789 error -101
```

Solution

The node needs to be drained before reboot. After the successful drain, the node can be rebooted as usual.

Because `kubectl drain` command automatically marks the node as unschedulable (`kubectl cordon` effect), the node needs to be uncordoned once it’s back online.

Drain the node:

```
kubectl drain <node-name> --ignore-daemonsets --delete-local-data
```

Uncordon the node:

```
kubectl uncordon <node-name>
```

Rook Agent modprobe exec format error

Symptoms

- PersistentVolumes from Ceph fail/timeout to mount
- Rook Agent logs contain `modinfo: ERROR: could not get modinfo from 'rbd': Exec format error` lines

Solution

If it is feasible to upgrade your kernel, you should upgrade to `4.x` , even better is

>= `4.7` due to a feature for CephFS added to the kernel.

If you are unable to upgrade the kernel, you need to go to each host that will consume storage and run:

```
1. modprobe rbd
```

This command inserts the `rbd` module into the kernel.

To persist this fix, you need to add the `rbd` kernel module to either `/etc/modprobe.d/` or `/etc/modules-load.d/`. For both paths create a file called `rbd.conf` with the following content:

```
1. rbd
```

Now when a host is restarted, the module should be loaded automatically.

Rook Agent rbd module missing error

Symptoms

- Rook Agent in `Error` or `CrashLoopBackOff` status when deploying the Rook operator with `kubectl create -f operator.yaml`:

```
1. $kubectl -n rook-ceph get pod
2. NAME                                READY     STATUS    RESTARTS   AGE
3. rook-ceph-agent-gfrm5               0/1       Error     0          14s
4. rook-ceph-operator-5f4866946-vmtff  1/1       Running   0          23s
5. rook-discover-qhx6c                 1/1       Running   0          14s
```

- Rook Agent logs contain below messages:

```
1. 2018-08-10 09:09:09.461798 I | exec: Running command: cat /lib/modules/4.15.2/modules.builtin
2. 2018-08-10 09:09:09.473858 I | exec: Running command: modinfo -F parm rbd
   2018-08-10 09:09:09.477215 N | ceph-volumeattacher: failed rbd single_major check, assuming it's unsupported:
   failed to check for rbd module single_major param: Failed to complete 'check kmod param': exit status 1.
3. modinfo: ERROR: Module rbd not found.
4. 2018-08-10 09:09:09.477239 I | exec: Running command: modprobe rbd
5. 2018-08-10 09:09:09.480353 I | modprobe rbd: modprobe: FATAL: Module rbd not found.
   2018-08-10 09:09:09.480452 N | ceph-volumeattacher: failed to load kernel module rbd: failed to load kernel
6. module rbd: Failed to complete 'modprobe rbd': exit status 1.
   failed to run rook ceph agent. failed to create volume manager: failed to load kernel module rbd: Failed to
7. complete 'modprobe rbd': exit status 1.
```

Solution

From the log message of Agent, we can see that the `rbd` kernel module is not

available in the current system, neither as a builtin nor a loadable external kernel module.

In this case, you have to re-configure and build a new kernel to address this issue, there're two options:

- Re-configure your kernel to make sure the `CONFIG_BLK_DEV_RBD=y` in the `.config` file, then build the kernel.
- Re-configure your kernel to make sure the `CONFIG_BLK_DEV_RBD=m` in the `.config` file, then build the kernel.

Rebooting the system to use the new kernel, this issue should be fixed: the Agent will be in normal `running` status if everything was done correctly.

Using multiple shared filesystem (CephFS) is attempted on a kernel version older than 4.7

Symptoms

- More than one shared filesystem (CephFS) has been created in the cluster
- A pod attempts to mount any other shared filesystem besides the **first** one that was created
- The pod incorrectly gets the first filesystem mounted instead of the intended filesystem

Solution

The only solution to this problem is to upgrade your kernel to `4.7` or higher. This is due to a mount flag added in the kernel version `4.7` which allows to chose the filesystem by name.

For additional info on the kernel version requirement for multiple shared filesystems (CephFS), see Filesystem - Kernel version requirement.

Activate log to file for a particular Ceph daemon

They are cases where looking at Kubernetes logs is not enough for diverse reasons, but just to name a few:

- not everyone is familiar for Kubernetes logging and expects to find logs in traditional directories
- logs get eaten (buffer limit from the log engine) and thus not requestable from Kubernetes

So for each daemon, `dataDirHostPath` is used to store logs, if logging is activated.

Rook will bindmount `dataDirHostPath` for every pod. As of Ceph Nautilus 14.2.1, it is possible to enable logging for a particular daemon on the fly. Let's say you want to enable logging for `mon.a`, but only for this daemon. Using the toolbox or from inside the operator run:

```
1. ceph config daemon mon.a log_to_file true
```

This will activate logging on the filesystem, you will be able to find logs in `dataDirHostPath/$NAMESPACE/log`, so typically this would mean `/var/lib/rook/rook-ceph/log`. You don't need to restart the pod, the effect will be immediate.

To disable the logging on file, simply set `log_to_file` to `false`.

For Ceph Luminous/Mimic releases, `mon_cluster_log_file` and `cluster_log_file` can be set to `/var/log/ceph/XXXX` in the config override ConfigMap to enable logging. See the (Advanced Documentation)[Documentation/advanced-configuration.md#kubernetes] for information about how to use the config override ConfigMap.

For Ceph Luminous/Mimic releases, `mon_cluster_log_file` and `cluster_log_file` can be set to `/var/log/ceph/XXXX` in the config override ConfigMap to enable logging. See the Advanced Documentation for information about how to use the config override ConfigMap.

Flex storage class versus Ceph CSI storage class

Since Rook 1.1, Ceph CSI has become stable and moving forward is the ultimate replacement over the Flex driver. However, not all Flex storage classes are available through Ceph CSI since it's basically catching up on features. Ceph CSI in its 1.2 version (with Rook 1.1) does not support the Erasure coded pools storage class.

So, if you are looking at using such storage class you should enable the Flex driver by setting `ROOK_ENABLE_FLEX_DRIVER: true` in your `operator.yaml`. Also, if you are in the need of specific features and wonder if CSI is capable of handling them, you should read the ceph-csi support matrix.

A worker node using RBD devices hangs up

Symptoms

- There is no progress on I/O from/to one of RBD devices (`/dev/rbd*` or `/dev/nbd*`).
- After that, the whole worker node hangs up.

Investigation

This hapens when the following conditions are satisfied.

- The problematic RBD device and the corresponding OSDs are co-located.
- There is an XFS filesystem on top of this device.

In addition, when this problem happens, you can see the following messages in `dmesg`.

```
# dmesg
...
[51717.039319] INFO: task kworker/2:1:5938 blocked for more than 120 seconds.
[51717.039361]       Not tainted 4.15.0-72-generic #81-Ubuntu
[51717.039388] "echo 0 > /proc/sys/kernel/hung_task_timeout_secs" disables this message.
...
```

It's so-called `hung_task` problem and means that there is a deadlock in the kernel. For more detail, please refer to the corresponding issue comment.

Solution

This problem will be solve by the following two fixes.

- Linux kernel: A minor feature that is introduced by this commit. It will be included in Linux v5.6.
- Ceph: A fix that uses the above-mentioned kernel's feature. The Ceph community will probably discuss this fix after releasing Linux v5.6.

You can bypass this problem by using ext4 or any other filesystems rather than XFS. Filesystem type can be specified with `csi.storage.k8s.io/fstype` in StorageClass resource.

Too few PGs per OSD warning is shown

Symptoms

- `ceph status` shows "too few PGs per OSD" warning as follows.

```
# ceph status
  cluster:
    id:     fd06d7c3-5c5c-45ca-bdea-1cf26b783065
    health: HEALTH_WARN
            too few PGs per OSD (16 < min 30)
```

Solution

The meaning of this warning is written in the document. However, in many cases it is benign. For more information, please see the blog entry. Please refer to Configuring Pools if you want to know the proper `pg_num` of pools and change these values.

Ceph OSD Management

Ceph Object Storage Daemons (OSDs) are the heart and soul of the Ceph storage platform. Each OSD manages a local device and together they provide the distributed storage. Rook will automate creation and management of OSDs to hide the complexity based on the desired state in the CephCluster CR as much as possible. This guide will walk through some of the scenarios to configure OSDs where more configuration may be required.

OSD Health

The rook-ceph-tools pod provides a simple environment to run Ceph tools. The `ceph` commands mentioned in this document should be run from the toolbox.

Once the is created, connect to the pod to execute the `ceph` commands to analyze the health of the cluster, in particular the OSDs and placement groups (PGs). Some common commands to analyze OSDs include:

```
ceph status
ceph osd tree
ceph osd status
ceph osd df
ceph osd utilization
```

```
kubectl -n rook-ceph exec -it $(kubectl -n rook-ceph get pod -l "app=rook-ceph-tools" -o
jsonpath='{.items[0].metadata.name}') bash
```

Add an OSD

The QuickStart Guide will provide the basic steps to create a cluster and start some OSDs. For more details on the OSD settings also see the Cluster CRD documentation. If you are not seeing OSDs created, see the Ceph Troubleshooting Guide.

To add more OSDs, Rook will automatically watch for new nodes and devices being added to your cluster. If they match the filters or other settings in the `storage` section of the cluster CR, the operator will create new OSDs.

Add an OSD on a PVC

In more dynamic environments where storage can be dynamically provisioned with a raw block storage provider, the OSDs can be backed by PVCs. See the `storageClassDeviceSets` documentation in the Cluster CRD topic.

To add more OSDs, you can either increase the `count` of the OSDs in an existing device set or you can add more device sets to the cluster CR. The operator will then automatically create new OSDs according to the updated cluster CR.

Remove an OSD

Removal of OSDs is intentionally not automated. Rook's charter is to keep your data safe, not to delete it. If you are sure you need to remove OSDs, it can be done. We just want you to be in control of this action.

To remove an OSD due to a failed disk or other re-configuration, consider the following to ensure the health of the data through the removal process:

- Confirm you will have enough space on your cluster after removing your OSDs to properly handle the deletion
- Confirm the remaining OSDs and their placement groups (PGs) are healthy in order to handle the rebalancing of the data
- Do not remove too many OSDs at once
- Wait for rebalancing between removing multiple OSDs

If all the PGs are `active+clean` and there are no warnings about being low on space, this means the data is fully replicated and it is safe to proceed. If an OSD is failing, the PGs will not be perfectly clean and you will need to proceed anyway.

From the Toolbox

1. Determine the OSD ID for the OSD to be removed. The osd pod may be in an error state such as `CrashLoopBackoff` or the `ceph` commands in the toolbox may show which OSD is `down`.
2. Mark the OSD as `out` if not already marked as such by Ceph. This signals Ceph to start moving (backfilling) the data that was on that OSD to another OSD.
    - `ceph osd out osd.<ID>` (for example if the OSD ID is 23 this would be `ceph osd out osd.23`)
3. Wait for the data to finish backfilling to other OSDs.
    - `ceph status` will indicate the backfilling is done when all of the PGs are `active+clean`. If desired, it's safe to remove the disk after that.
4. Update your CephCluster CR such that the operator won't create an OSD on the device anymore. Depending on your CR settings, you may need to remove the device from the list or update the device filter. If you are using `useAllDevices: true`, no change to the CR is necessary.
5. Remove the OSD from the Ceph cluster
    - `ceph osd purge <ID> --yes-i-really-mean-it`
6. Verify the OSD is removed from the node in the CRUSH map
    - `ceph osd tree`

Remove the OSD Deployment

The operator can automatically remove OSD deployments that are considered “safe-to-destroy” by Ceph. After the steps above, the OSD will be considered safe to remove since the data has all been moved to other OSDs. But this will only be done automatically by the operator if you have this setting in the cluster CR:

```
removeOSDsIfOutAndSafeToRemove: true
```

1. Otherwise, you will need to delete the deployment directly:
    - `kubectl delete deployment -n rook-ceph rook-ceph-osd-<ID>`

Delete the underlying data

1. If you want to clean the device where the OSD was running, see in the instructions to wipe a disk on the Cleaning up a Cluster topic.

Replace an OSD

To replace a disk that has failed:

1. Run the steps in the previous section to Remove an OSD.
2. Replace the physical device and verify the new device is attached.
3. Check if your cluster CR will find the new device. If you are using `useAllDevices: true` you can skip this step. If your cluster CR lists individual devices or uses a device filter you may need to update the CR.
4. The operator ideally will automatically create the new OSD within a few minutes of adding the new device or updating the CR. If you don’t see a new OSD automatically created, restart the operator (by deleting the operator pod) to trigger the OSD creation.
5. Verify if the OSD is created on the node by running `ceph osd tree` from the toolbox.

Note that the OSD might have a different ID than the previous OSD that was replaced.

Remove an OSD from a PVC

If you have installed your OSDs on top of PVCs and you desire to reduce the size of your cluster by removing OSDs:

1. Shrink the number of OSDs in the `storageClassDeviceSet` in the CephCluster CR.
    - `kubectl -n rook-ceph edit cephcluster rook-ceph`
    - Reduce the `count` of the OSDs to the desired number. Rook will not take any action to automatically remove the extra OSD(s), but will effectively stop managing the orphaned OSD.
2. Identify the orphaned PVC that belongs to the orphaned OSD.
    - The orphaned PVC will have the highest index among the PVCs for the device set.
    - `kubectl -n rook-ceph get pvc -l ceph.rook.io/DeviceSet=<deviceSet>`

    - For example if the device set is named `set1` and the `count` was reduced from `3` to `2`, the orphaned PVC would have the index `2` and might be named `set1-2-data-vbwcf`

3. Identify the orphaned OSD.
    - The OSD assigned to the PVC can be found in the labels on the PVC
    - `kubectl -n rook-ceph get pod -l ceph.rook.io/pvc=<orphaned-pvc> -o yaml | grep ceph-osd-id`
    - For example, this might return: `ceph-osd-id: "0"`
4. Now proceed with the steps in the section above to Remove an OSD for the orphaned OSD ID.
5. If desired, delete the orphaned PVC after the OSD is removed.

OpenShift Common Issues

Enable Monitoring in the Storage Dashboard

OpenShift Console uses OpenShift Prometheus for monitoring and populating data in Storage Dashboard. Additional configuration is required to monitor the Ceph Cluster from the storage dashboard.

1. Change the monitoring namespace to `openshift-monitoring`

   Change the namespace of the RoleBinding `rook-ceph-metrics` from `rook-ceph` to `openshift-monitoring` for the `prometheus-k8s` ServiceAccount in rbac.yaml.

```
subjects:
- kind: ServiceAccount
  name: prometheus-k8s
  namespace: openshift-monitoring
```

1. Enable Ceph Cluster monitoring

   Follow ceph-monitoring/prometheus-alerts.

2. Set the required label on the namespace

   `$ oc label namespace rook-ceph "openshift.io/cluster-monitoring=true"`

Troubleshoot Monitoring Issues

> **Pre-req:** Switch to `rook-ceph` namespace with `oc project rook-ceph`

1. Ensure ceph-mgr pod is Running

   `$ oc get pods -l app=rook-ceph-mgr`

```
NAME            READY   STATUS    RESTARTS   AGE
rook-ceph-mgr   1/1     Running   0          14h
```

2. Ensure service monitor is present

   `$ oc get servicemonitor rook-ceph-mgr`

```
NAME                          AGE
rook-ceph-mgr                 14h
```

3. Ensure prometheus rules are present

```
oc get prometheusrules -l prometheus=rook-prometheus
```

```
NAME                    AGE
prometheus-ceph-rules   14h
```

Ceph Tools

Rook provides a number of tools to help you manage your cluster.

- Common Issues: Common issues and their potential solutions
- Toolbox: A pod from which you can run all of the tools to troubleshoot the storage cluster
- Direct Tools: Run ceph commands to test directly mounting block and file storage
- Advanced Configuration: Tips and tricks for configuring for cluster
- Container Linux Update Operator: Configure the container linux update operator to manage updates to the nodes
- Disaster Recovery: In the worst case scenario if the ceph mons lose quorum, follow these steps to recover

Rook Toolbox

The Rook toolbox is a container with common tools used for rook debugging and testing. The toolbox is based on CentOS, so more tools of your choosing can be easily installed with `yum` .

The toolbox can be run in two modes:

1. Interactive: Start a toolbox pod where you can connect and execute Ceph commands from a shell
2. One-time job: Run a script with Ceph commands and collect the results from the job log

> Prerequisite: Before running the toolbox you should have a running Rook cluster deployed (see the Quickstart Guide).

Interactive Toolbox

The rook toolbox can run as a deployment in a Kubernetes cluster where you can connect and run arbitrary Ceph commands.

Save the tools spec as `toolbox.yaml` :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rook-ceph-tools
  namespace: rook-ceph
  labels:
    app: rook-ceph-tools
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rook-ceph-tools
  template:
    metadata:
      labels:
        app: rook-ceph-tools
    spec:
      dnsPolicy: ClusterFirstWithHostNet
      containers:
      - name: rook-ceph-tools
        image: rook/ceph:v1.4.1
        command: ["/tini"]
        args: ["-g", "--", "/usr/local/bin/toolbox.sh"]
        imagePullPolicy: IfNotPresent
        env:
```

```
          - name: ROOK_CEPH_USERNAME
            valueFrom:
              secretKeyRef:
                name: rook-ceph-mon
                key: ceph-username
          - name: ROOK_CEPH_SECRET
            valueFrom:
              secretKeyRef:
                name: rook-ceph-mon
                key: ceph-secret
        volumeMounts:
          - mountPath: /etc/ceph
            name: ceph-config
          - name: mon-endpoint-volume
            mountPath: /etc/rook
      volumes:
        - name: mon-endpoint-volume
          configMap:
            name: rook-ceph-mon-endpoints
            items:
            - key: data
              path: mon-endpoints
        - name: ceph-config
          emptyDir: {}
      tolerations:
        - key: "node.kubernetes.io/unreachable"
          operator: "Exists"
          effect: "NoExecute"
          tolerationSeconds: 5
```

Launch the rook-ceph-tools pod:

```
kubectl create -f toolbox.yaml
```

Wait for the toolbox pod to download its container and get to the `running` state:

```
kubectl -n rook-ceph get pod -l "app=rook-ceph-tools"
```

Once the rook-ceph-tools pod is running, you can connect to it with:

```
kubectl -n rook-ceph exec -it $(kubectl -n rook-ceph get pod -l "app=rook-ceph-tools" -o
jsonpath='{.items[0].metadata.name}') bash
```

All available tools in the toolbox are ready for your troubleshooting needs.

Example:

- `ceph status`
- `ceph osd status`
- `ceph df`

- `rados df`

When you are done with the toolbox, you can remove the deployment:

```
kubectl -n rook-ceph delete deployment rook-ceph-tools
```

Toolbox Job

If you want to run Ceph commands as a one-time operation and collect the results later from the logs, you can run a script as a Kubernetes Job. The toolbox job will run a script that is embedded in the job spec. The script has the full flexibility of a bash script.

In this example, the `ceph status` command is executed when the job is created.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: rook-ceph-toolbox-job
  namespace: rook-ceph
  labels:
    app: ceph-toolbox-job
spec:
  template:
    spec:
      initContainers:
      - name: config-init
        image: rook/ceph:v1.4.1
        command: ["/usr/local/bin/toolbox.sh"]
        args: ["--skip-watch"]
        imagePullPolicy: IfNotPresent
        env:
        - name: ROOK_CEPH_USERNAME
          valueFrom:
            secretKeyRef:
              name: rook-ceph-mon
              key: ceph-username
        - name: ROOK_CEPH_SECRET
          valueFrom:
            secretKeyRef:
              name: rook-ceph-mon
              key: ceph-secret
        volumeMounts:
        - mountPath: /etc/ceph
          name: ceph-config
        - name: mon-endpoint-volume
          mountPath: /etc/rook
      containers:
      - name: script
        image: rook/ceph:v1.4.1
```

```
        volumeMounts:
        - mountPath: /etc/ceph
          name: ceph-config
          readOnly: true
        command:
        - "bash"
        - "-c"
        - |
          # Modify this script to run any ceph, rbd, radosgw-admin, or other commands that could
          # be run in the toolbox pod. The output of the commands can be seen by getting the pod log.
          #
          # example: print the ceph status
          ceph status
      volumes:
      - name: mon-endpoint-volume
        configMap:
          name: rook-ceph-mon-endpoints
          items:
          - key: data
            path: mon-endpoints
      - name: ceph-config
        emptyDir: {}
      restartPolicy: Never
```

Create the toolbox job:

```
kubectl create -f toolbox-job.yaml
```

After the job completes, see the results of the script:

```
kubectl -n rook-ceph logs -l job-name=rook-ceph-toolbox-job
```

Direct Tools

Rook is designed with Kubernetes design principles from the ground up. This topic is going to escape the bounds of Kubernetes storage and show you how to use block and file storage directly from a pod without any of the Kubernetes magic. The purpose of this topic is to help you quickly test a new configuration, although it is not meant to be used in production. All of the benefits of Kubernetes storage including failover, detach, and attach will not be available. If your pod dies, your mount will die with it.

Start the Direct Mount Pod

To test mounting your Ceph volumes, start a pod with the necessary mounts. An example is provided in the examples test directory:

```
kubectl create -f cluster/examples/kubernetes/ceph/direct-mount.yaml
```

After the pod is started, connect to it like this:

```
kubectl -n rook-ceph get pod -l app=rook-direct-mount
kubectl -n rook-ceph exec -it <pod> bash
```

Block Storage Tools

After you have created a pool as described in the Block Storage topic, you can create a block image and mount it directly in a pod. This example will show how the Ceph rbd volume can be mounted in the direct mount pod.

Create the Direct Mount Pod.

Create a volume image (10MB):

```
rbd create replicapool/test --size 10
rbd info replicapool/test

# Disable the rbd features that are not in the kernel module
rbd feature disable replicapool/test fast-diff deep-flatten object-map
```

Map the block volume and format it and mount it:

```
# Map the rbd device. If the Direct Mount Pod was started with "hostNetwork: false" this hangs and you have to stop it with Ctrl-C,
# however the command still succeeds; see https://github.com/rook/rook/issues/2021
rbd map replicapool/test
```

```
# Find the device name, such as rbd0
lsblk | grep rbd

# Format the volume (only do this the first time or you will lose data)
mkfs.ext4 -m0 /dev/rbd0

# Mount the block device
mkdir /tmp/rook-volume
mount /dev/rbd0 /tmp/rook-volume
```

Write and read a file:

```
echo "Hello Rook" > /tmp/rook-volume/hello
cat /tmp/rook-volume/hello
```

Unmount the Block device

Unmount the volume and unmap the kernel device:

```
umount /tmp/rook-volume
rbd unmap /dev/rbd0
```

Shared Filesystem Tools

After you have created a filesystem as described in the Shared Filesystem topic, you can mount the filesystem from multiple pods. The the other topic you may have mounted the filesystem already in the registry pod. Now we will mount the same filesystem in the Direct Mount pod. This is just a simple way to validate the Ceph filesystem and is not recommended for production Kubernetes pods.

Follow Direct Mount Pod to start a pod with the necessary mounts and then proceed with the following commands after connecting to the pod.

```
# Create the directory
mkdir /tmp/registry

# Detect the mon endpoints and the user secret for the connection
mon_endpoints=$(grep mon_host /etc/ceph/ceph.conf | awk '{print $3}')
my_secret=$(grep key /etc/ceph/keyring | awk '{print $3}')

# Mount the filesystem
mount -t ceph -o mds_namespace=myfs,name=admin,secret=$my_secret $mon_endpoints:/ /tmp/registry

# See your mounted filesystem
df -h
```

Now you should have a mounted filesystem. If you have pushed images to the registry you will see a directory called `docker` .

```
ls /tmp/registry
```

Try writing and reading a file to the shared filesystem.

```
echo "Hello Rook" > /tmp/registry/hello
cat /tmp/registry/hello

# delete the file when you're done
rm -f /tmp/registry/hello
```

Unmount the Filesystem

To unmount the shared filesystem from the Direct Mount Pod:

```
umount /tmp/registry
rmdir /tmp/registry
```

No data will be deleted by unmounting the filesystem.

Advanced Configuration

These examples show how to perform advanced configuration tasks on your Rook storage cluster.

Prerequisites

Most of the examples make use of the `ceph` client command. A quick way to use the Ceph client suite is from a Rook Toolbox container.

The Kubernetes based examples assume Rook OSD pods are in the `rook-ceph` namespace. If you run them in a different namespace, modify `kubectl -n rook-ceph [...]` to fit your situation.

Use custom Ceph user and secret for mounting

> **NOTE**: For extensive info about creating Ceph users, consult the Ceph documentation: http://docs.ceph.com/docs/mimic/rados/operations/user-management/#add-a-user.

Using a custom Ceph user and secret can be done for filesystem and block storage.

Create a custom user in Ceph with read-write access in the `/bar` directory on CephFS (For Ceph Mimic or newer, use `data=POOL_NAME` instead of `pool=POOL_NAME`):

```
    ceph auth get-or-create-key client.user1 mon 'allow r' osd 'allow rw tag cephfs pool=YOUR_FS_DATA_POOL' mds
1. 'allow r, allow rw path=/bar'
```

The command will return a Ceph secret key, this key should be added as a secret in Kubernetes like this:

```
1. kubectl create secret generic ceph-user1-secret --from-literal=key=YOUR_CEPH_KEY
```

> **NOTE**: This secret with the same name must be created in each namespace where the StorageClass will be used.

In addition to this Secret you must create a RoleBinding to allow the Rook Ceph agent to get the secret from each namespace. The RoleBinding is optional if you are using a ClusterRoleBinding for the Rook Ceph agent secret access. A ClusterRole which contains the permissions which are needed and used for the Bindings are shown as an example after the next step.

On a StorageClass `parameters` and/or flexvolume Volume entry `options` set the following options:

```
1. mountUser: user1
2. mountSecret: ceph-user1-secret
```

If you want the Rook Ceph agent to require a `mountUser` and `mountSecret` to be set in StorageClasses using Rook, you must set the environment variable `AGENT_MOUNT_SECURITY_MODE` to `Restricted` on the Rook Ceph operator Deployment.

For more information on using the Ceph feature to limit access to CephFS paths, see Ceph Documentation - Path Restriction.

ClusterRole

> **NOTE**: When you are using the Helm chart to install the Rook Ceph operator and have set `mountSecurityMode` to e.g., `Restricted`, then the below ClusterRole has already been created for you.

This ClusterRole is needed no matter if you want to use a RoleBinding per namespace or a ClusterRoleBinding.

```
1. apiVersion: rbac.authorization.k8s.io/v1beta1
2. kind: ClusterRole
3. metadata:
4.   name: rook-ceph-agent-mount
5.   labels:
6.     operator: rook
7.     storage-backend: ceph
8. rules:
9. - apiGroups:
10.   - ""
11.   resources:
12.   - secrets
13.   verbs:
14.   - get
```

RoleBinding

> **NOTE**: You either need a RoleBinding in each namespace in which a mount secret resides in or create a ClusterRoleBinding with which the Rook Ceph agent has access to Kubernetes secrets in all namespaces.

Create the RoleBinding shown here in each namespace the Rook Ceph agent should read secrets for mounting. The RoleBinding `subjects`' `namespace` must be the one the Rook Ceph agent runs in (default `rook-ceph` for version 1.0 and newer. The default namespace in previous versions was `rook-ceph-system`).

Replace `namespace: name-of-namespace-with-mountsecret` according to the name of all namespaces a `mountSecret` can be in.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: rook-ceph-agent-mount
  namespace: name-of-namespace-with-mountsecret
  labels:
    operator: rook
    storage-backend: ceph
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: rook-ceph-agent-mount
subjects:
- kind: ServiceAccount
  name: rook-ceph-system
  namespace: rook-ceph
```

ClusterRoleBinding

This ClusterRoleBinding only needs to be created once, as it covers the whole cluster.

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: rook-ceph-agent-mount
  labels:
    operator: rook
    storage-backend: ceph
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: rook-ceph-agent-mount
subjects:
- kind: ServiceAccount
  name: rook-ceph-system
  namespace: rook-ceph
```

Log Collection

All Rook logs can be collected in a Kubernetes environment with the following command:

```
for p in $(kubectl -n rook-ceph get pods -o jsonpath='{.items[*].metadata.name}')
do
    for c in $(kubectl -n rook-ceph get pod ${p} -o jsonpath='{.spec.containers[*].name}')
    do
        echo "BEGIN logs from pod: ${p} ${c}"
        kubectl -n rook-ceph logs -c ${c} ${p}
        echo "END logs from pod: ${p} ${c}"
    done
done
```

This gets the logs for every container in every Rook pod and then compresses them into a `.gz` archive for easy sharing. Note that instead of `gzip`, you could instead pipe to `less` or to a single text file.

OSD Information

Keeping track of OSDs and their underlying storage devices can be difficult. The following scripts will clear things up quickly.

Kubernetes

```
# Get OSD Pods
# This uses the example/default cluster name "rook"
OSD_PODS=$(kubectl get pods --all-namespaces -l \
  app=rook-ceph-osd,rook_cluster=rook-ceph -o jsonpath='{.items[*].metadata.name}')

# Find node and drive associations from OSD pods
for pod in $(echo ${OSD_PODS})
do
 echo "Pod:  ${pod}"
 echo "Node: $(kubectl -n rook-ceph get pod ${pod} -o jsonpath='{.spec.nodeName}')"
 kubectl -n rook-ceph exec ${pod} -- sh -c '\
  for i in /var/lib/ceph/osd/ceph-*; do
    [ -f ${i}/ready ] || continue
    echo -ne "-$(basename ${i}) "
    echo $(lsblk -n -o NAME,SIZE ${i}/block 2> /dev/null || \
    findmnt -n -v -o SOURCE,SIZE -T ${i}) $(cat ${i}/type)
  done | sort -V
  echo'
done
```

The output should look something like this.

```
Pod:  osd-m2fz2
Node: node1.zbrbdl
-osd0  sda3  557.3G  bluestore
-osd1  sdf3  110.2G  bluestore
-osd2  sdd3  277.8G  bluestore
-osd3  sdb3  557.3G  bluestore
```

```
-osd4  sde3  464.2G  bluestore
-osd5  sdc3  557.3G  bluestore

Pod:  osd-nxxnq
Node: node3.zbrbdl
-osd6   sda3  110.7G  bluestore
-osd17  sdd3  1.8T    bluestore
-osd18  sdb3  231.8G  bluestore
-osd19  sdc3  231.8G  bluestore

Pod:  osd-tww1h
Node: node2.zbrbdl
-osd7   sdc3  464.2G  bluestore
-osd8   sdj3  557.3G  bluestore
-osd9   sdf3  66.7G   bluestore
-osd10  sdd3  464.2G  bluestore
-osd11  sdb3  147.4G  bluestore
-osd12  sdi3  557.3G  bluestore
-osd13  sdk3  557.3G  bluestore
-osd14  sde3  66.7G   bluestore
-osd15  sda3  110.2G  bluestore
-osd16  sdh3  135.1G  bluestore
```

Separate Storage Groups

> **DEPRECATED**: Instead of manually needing to set this, the `deviceClass` property can be used on Pool structures in `CephBlockPool`, `CephFilesystem` and `CephObjectStore` CRD objects.

By default Rook/Ceph puts all storage under one replication rule in the CRUSH Map which provides the maximum amount of storage capacity for a cluster. If you would like to use different storage endpoints for different purposes, you'll have to create separate storage groups.

In the following example we will separate SSD drives from spindle-based drives, a common practice for those looking to target certain workloads onto faster (database) or slower (file archive) storage.

Configuring Pools

Placement Group Sizing

> **NOTE**: Since Ceph Nautilus (v14.x), you can use the Ceph MGR `pg_autoscaler` module to auto scale the PGs as needed. If you want to enable this feature, please refer to Default PG and PGP counts.

The general rules for deciding how many PGs your pool(s) should contain is:

- Less than 5 OSDs set pg_num to 128
- Between 5 and 10 OSDs set pg_num to 512
- Between 10 and 50 OSDs set pg_num to 1024

If you have more than 50 OSDs, you need to understand the tradeoffs and how to calculate the pg_num value by yourself. For calculating pg_num yourself please make use of the pgcalc tool.

If you're already using a pool it is generally safe to increase its PG count on-the-fly. Decreasing the PG count is not recommended on a pool that is in use. The safest way to decrease the PG count is to back-up the data, delete the pool, and recreate it. With backups you can try a few potentially unsafe tricks for live pools, documented here.

Setting PG Count

Be sure to read the placement group sizing section before changing the number of PGs.

```
1. # Set the number of PGs in the rbd pool to 512
2. ceph osd pool set rbd pg_num 512
```

Custom ceph.conf Settings

> **WARNING**: The advised method for controlling Ceph configuration is to manually use the Ceph CLI or the Ceph dashboard because this offers the most flexibility. It is highly recommended that this only be used when absolutely necessary and that the `config` be reset to an empty string if/when the configurations are no longer necessary. Configurations in the config file will make the Ceph cluster less configurable from the CLI and dashboard and may make future tuning or debugging difficult.

Setting configs via Ceph's CLI requires that at least one mon be available for the configs to be set, and setting configs via dashboard requires at least one mgr to be available. Ceph may also have a small number of very advanced settings that aren't able to be modified easily via CLI or dashboard. In order to set configurations before monitors are available or to set problematic configuration settings, the `rook-config-override` ConfigMap exists, and the `config` field can be set with the contents of a `ceph.conf` file. The contents will be propagated to all mon, mgr, OSD, MDS, and RGW daemons as an `/etc/ceph/ceph.conf` file.

> **WARNING**: Rook performs no validation on the config, so the validity of the settings is the user's responsibility.

If the `rook-config-override` ConfigMap is created before the cluster is started, the Ceph daemons will automatically pick up the settings. If you add the settings to the ConfigMap after the cluster has been initialized, each daemon will need to be restarted where you want the settings applied:

- mons: ensure all three mons are online and healthy before restarting each mon pod, one at a time.
- mgrs: the pods are stateless and can be restarted as needed, but note that this will disrupt the Ceph dashboard during restart.
- OSDs: restart your the pods by deleting them, one at a time, and running `ceph -s` between each restart to ensure the cluster goes back to "active/clean" state.

- RGW: the pods are stateless and can be restarted as needed.
- MDS: the pods are stateless and can be restarted as needed.

After the pod restart, the new settings should be in effect. Note that if the ConfigMap in the Ceph cluster's namespace is created before the cluster is created, the daemons will pick up the settings at first launch.

Example

In this example we will set the default pool `size` to two, and tell OSD daemons not to change the weight of OSDs on startup.

> **WARNING**: Modify Ceph settings carefully. You are leaving the sandbox tested by Rook. Changing the settings could result in unhealthy daemons or even data loss if used incorrectly.

When the Rook Operator creates a cluster, a placeholder ConfigMap is created that will allow you to override Ceph configuration settings. When the daemon pods are started, the settings specified in this ConfigMap will be merged with the default settings generated by Rook.

The default override settings are blank. Cutting out the extraneous properties, we would see the following defaults after creating a cluster:

```
$ kubectl -n rook-ceph get ConfigMap rook-config-override -o yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: rook-config-override
  namespace: rook-ceph
data:
  config: ""
```

To apply your desired configuration, you will need to update this ConfigMap. The next time the daemon pod(s) start, they will use the updated configs.

```
kubectl -n rook-ceph edit configmap rook-config-override
```

Modify the settings and save. Each line you add should be indented from the `config` property as such:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: rook-config-override
  namespace: rook-ceph
data:
  config: |
    [global]
    osd crush update on start = false
```

```
    osd pool default size = 2
```

OSD CRUSH Settings

A useful view of the CRUSH Map is generated with the following command:

```
ceph osd tree
```

In this section we will be tweaking some of the values seen in the output.

OSD Weight

The CRUSH weight controls the ratio of data that should be distributed to each OSD. This also means a higher or lower amount of disk I/O operations for an OSD with higher/lower weight, respectively.

By default OSDs get a weight relative to their storage capacity, which maximizes overall cluster capacity by filling all drives at the same rate, even if drive sizes vary. This should work for most use-cases, but the following situations could warrant weight changes:

- Your cluster has some relatively slow OSDs or nodes. Lowering their weight can reduce the impact of this bottleneck.
- You’re using bluestore drives provisioned with Rook v0.3.1 or older. In this case you may notice OSD weights did not get set relative to their storage capacity. Changing the weight can fix this and maximize cluster capacity.

This example sets the weight of osd.0 which is 600GiB

```
ceph osd crush reweight osd.0 .600
```

OSD Primary Affinity

When pools are set with a size setting greater than one, data is replicated between nodes and OSDs. For every chunk of data a Primary OSD is selected to be used for reading that data to be sent to clients. You can control how likely it is for an OSD to become a Primary using the Primary Affinity setting. This is similar to the OSD weight setting, except it only affects reads on the storage device, not capacity or writes.

In this example we will make sure `osd.0` is only selected as Primary if all other OSDs holding replica data are unavailable:

```
ceph osd primary-affinity osd.0 0
```

OSD Dedicated Network

It is possible to configure ceph to leverage a dedicated network for the OSDs to communicate across. A useful overview is the CEPH Networks section of the Ceph documentation. If you declare a cluster network, OSDs will route heartbeat, object replication and recovery traffic over the cluster network. This may improve performance compared to using a single network.

Two changes are necessary to the configuration to enable this capability:

Use hostNetwork in the rook ceph cluster configuration

Enable the `hostNetwork` setting in the Ceph Cluster CRD configuration. For example,

```
1.  network:
2.    hostNetwork: true
```

> IMPORTANT: Changing this setting is not supported in a running Rook cluster. Host networking should be configured when the cluster is first created.

Define the subnets to use for public and private OSD networks

Edit the `rook-config-override` configmap to define the custom network configuration:

```
1. kubectl -n rook-ceph edit configmap rook-config-override
```

In the editor, add a custom configuration to instruct ceph which subnet is the public network and which subnet is the private network. For example:

```
1. apiVersion: v1
2. data:
3.   config: |
4.     [global]
5.     public network =  10.0.7.0/24
6.     cluster network = 10.0.10.0/24
7.     public addr = ""
8.     cluster addr = ""
```

After applying the updated rook-config-override configmap, it will be necessary to restart the OSDs by deleting the OSD pods in order to apply the change. Restart the OSD pods by deleting them, one at a time, and running ceph -s between each restart to ensure the cluster goes back to “active/clean” state.

Phantom OSD Removal

If you have OSDs in which are not showing any disks, you can remove those “Phantom OSDs” by following the instructions below. To check for “Phantom OSDs”, you can run:

```
ceph osd tree
```

An example output looks like this:

```
ID  CLASS WEIGHT  TYPE NAME STATUS REWEIGHT PRI-AFF
 -1       57.38062 root default
-13        7.17258     host node1.example.com
  2   hdd  3.61859         osd.2               up  1.00000 1.00000
 -7              0     host node2.example.com   down    0    1.00000
```

The host `node2.example.com` in the output has no disks, so it is most likely a “Phantom OSD”.

Now to remove it, use the ID in the first column of the output and replace `<ID>` with it. In the example output above the ID would be `-7`. The commands are:

```
ceph osd out <ID>
ceph osd crush remove osd.<ID>
ceph auth del osd.<ID>
ceph osd rm <ID>
```

To recheck that the Phantom OSD was removed, re-run the following command and check if the OSD with the ID doesn’t show up anymore:

```
ceph osd tree
```

Change Failure Domain

In Rook, it is now possible to indicate how the default CRUSH failure domain rule must be configured in order to ensure that replicas or erasure code shards are separated across hosts, and a single host failure does not affect availability. For instance, this is an example manifest of a block pool named `replicapool` configured with a `failureDomain` set to `osd`:

```
apiVersion: ceph.rook.io/v1
kind: CephBlockPool
metadata:
  name: replicapool
  namespace: rook
spec:
  # The failure domain will spread the replicas of the data across different failure zones
```

```
    failureDomain: osd
    ...
```

However, due to several reasons, we may need to change such failure domain to its other value: `host`. Unfortunately, changing it directly in the YAML manifest is not currently handled by Rook, so we need to perform the change directly using Ceph commands using the Rook tools pod, for instance:

```
$ ceph osd pool get replicapool crush_rule
crush_rule: replicapool

$ceph osd crush rule create-replicated replicapool_host_rule default host
```

Notice that the suffix `host_rule` in the name of the rule is just for clearness about the type of rule we are creating here, and can be anything else as long as it is different from the existing one. Once the new rule has been created, we simply apply it to our block pool:

```
ceph osd pool set replicapool crush_rule replicapool_host_rule
```

And validate that it has been actually applied properly:

```
$ ceph osd pool get replicapool crush_rule
crush_rule: replicapool_host_rule
```

If the cluster's health was `HEALTH_OK` when we performed this change, immediately, the new rule is applied to the cluster transparently without service disruption.

Exactly the same approach can be used to change from `host` back to `osd`.

Using the Container Linux Update Operator with Rook

When you are using Container Linux (CoreOS) and have the update engine enabled, it could be that a node reboots quickly after another not leaving enough time for the Rook cluster to rebuild. The Container Linux Update Operator is the solution for this, you can block your nodes to reboot until the Ceph cluster is healthy.

Prerequisites

- An operational Container Linux Kubernetes cluster (Successfully tested with 1.8.4)
- A working rook cluster
- The update-engine.service systemd unit on each machine should be unmasked, enabled and started in systemd
- The locksmithd.service systemd unit on each machine should be masked and stopped in systemd

Start the update operator

Proper reading of the README on the Container Linux Update Operator is necessary. Clone the repo and go in the `examples` directory.

Look for the file named `update-operator.yaml` and update the `command` part of the container from:

```
1. command:
2. - "/bin/update-operator"
```

to:

```
1. command:
2. - "/bin/update-operator"
3. - "--before-reboot-annotations"
4. - "ceph-before-reboot-check"
5. - "--after-reboot-annotations"
6. - "ceph-after-reboot-check"
```

You can also add the `-v 6` argument for more extensive logging.

Now create the update-operator by invoking following commands:

```
1. kubectl create -f namespace.yaml
2. kubectl create -f cluster-role.yaml
```

```
kubectl create -f cluster-role-binding.yaml
kubectl create -f update-operator.yaml
kubectl create -f update-agent.yaml
```

These files create a new namespace `reboot-coordinator`, configured to listen for the node annotation `ceph-reboot-check`. Now you can create both files in the `cluster/examples/coreos` folder, here's a short description of what each file does:

- `rbac.yaml`: This file contains the necessary RBAC settings.
- `ceph-after-reboot-script.yaml`: This file creates a `ConfigMap` containing a bash script which will be mounted in the `rook-toolbox` image as executable file.
- `ceph-before-reboot-script.yaml`: This file creates a `ConfigMap` containing a bash script which will be mounted in the `rook-toolbox` image as executable file.
- `before-reboot-daemonset.yaml`: This file creates a `DaemonSet` which waits for a node being labeled `before-reboot=true`, runs and checks the Ceph status. If all is correct, it annotates the node with `ceph-before-reboot-check=true`.
- `after-reboot-daemonset.yaml`: This file creates a `DaemonSet` which waits for a node being labeled `after-reboot=true`, runs and unsets the `noout` option for the ceph OSDs. If all is correct, it annotates the node with `ceph-after-reboot-check=true`.

The node annotation `ceph-no-noout=true` can be used to avoid `ceph-before-reboot-check` from setting the OSD `noout` flag. This annotation should only be used when deleting a node from a cluster, this way the cluster starts rebalancing immediately, not waiting for the node to come back up.

```
kubectl create -f rbac.yaml
kubectl create -f ceph-after-reboot-script.yaml
kubectl create -f ceph-before-reboot-script.yaml
kubectl create -f before-reboot-daemonset.yaml
kubectl create -f after-reboot-daemonset.yaml
```

Destroy the update operator

To destroy all elements created in this file, run:

```
kubectl delete -f before-reboot-daemonset.yaml
kubectl delete -f after-reboot-daemonset.yaml
kubectl delete -f ceph-after-reboot-script.yaml
kubectl delete -f ceph-before-reboot-script.yaml
kubectl delete -f rbac.yaml
```

Then you may safely delete the update operator itself: From the directory of the Container Linux Update Operator you cloned earlier, go again into the `examples` folder and run following commands:

```
kubectl delete -f update-agent.yaml
kubectl delete -f update-operator.yaml
```

```
kubectl delete -f cluster-role-binding.yaml
kubectl delete -f cluster-role.yaml
kubectl delete -f namespace.yaml
```

Disaster Recovery

Restoring Mon Quorum

Under extenuating circumstances, the mons may lose quorum. If the mons cannot form quorum again, there is a manual procedure to get the quorum going again. The only requirement is that at least one mon is still healthy. The following steps will remove the unhealthy mons from quorum and allow you to form a quorum again with a single mon, then grow the quorum back to the original size.

For example, if you have three mons and lose quorum, you will need to remove the two bad mons from quorum, notify the good mon that it is the only mon in quorum, and then restart the good mon.

Stop the operator

First, stop the operator so it will not try to failover the mons while we are modifying the monmap

```
1. kubectl -n rook-ceph scale deployment rook-ceph-operator --replicas=0
```

Inject a new monmap

> **WARNING**: Injecting a monmap must be done very carefully. If run incorrectly, your cluster could be permanently destroyed.

The Ceph monmap keeps track of the mon quorum. We will update the monmap to only contain the healthy mon. In this example, the healthy mon is `rook-ceph-mon-b`, while the unhealthy mons are `rook-ceph-mon-a` and `rook-ceph-mon-c`.

Take a backup of the current `rook-ceph-mon-b` Deployment:

```
1. kubectl -n rook-ceph get deployment rook-ceph-mon-b -o yaml > rook-ceph-mon-b-deployment.yaml
```

Open the file and copy the `command` and `args` from the `mon` container (see `containers` list). This is needed for the monmap changes. Cleanup the copied `command` and `args` fields to form a pastable command. Example:

The following parts of the `mon` container:

```
1. [...]
2.   containers:
3.   - args:
4.     - --fsid=41a537f2-f282-428e-989f-a9e07be32e47
5.     - --keyring=/etc/ceph/keyring-store/keyring
```

```
    - --log-to-stderr=true
    - --err-to-stderr=true
    - --mon-cluster-log-to-stderr=true
    - '--log-stderr-prefix=debug '
    - --default-log-to-file=false
    - --default-mon-cluster-log-to-file=false
    - --mon-host=$(ROOK_CEPH_MON_HOST)
    - --mon-initial-members=$(ROOK_CEPH_MON_INITIAL_MEMBERS)
    - --id=b
    - --setuser=ceph
    - --setgroup=ceph
    - --foreground
    - --public-addr=10.100.13.242
    - --setuser-match-path=/var/lib/ceph/mon/ceph-b/store.db
    - --public-bind-addr=$(ROOK_POD_IP)
    command:
    - ceph-mon
[...]
```

Should be made into a command like this: (**do not copy the example command!**)

```
ceph-mon \
    --fsid=41a537f2-f282-428e-989f-a9e07be32e47 \
    --keyring=/etc/ceph/keyring-store/keyring \
    --log-to-stderr=true \
    --err-to-stderr=true \
    --mon-cluster-log-to-stderr=true \
    --log-stderr-prefix=debug \
    --default-log-to-file=false \
    --default-mon-cluster-log-to-file=false \
    --mon-host=$(ROOK_CEPH_MON_HOST) \
    --mon-initial-members=$(ROOK_CEPH_MON_INITIAL_MEMBERS) \
    --id=b \
    --setuser=ceph \
    --setgroup=ceph \
    --foreground \
    --public-addr=10.100.13.242 \
    --setuser-match-path=/var/lib/ceph/mon/ceph-b/store.db \
    --public-bind-addr=$(ROOK_POD_IP)
```

(be sure to remove the single quotes around the `--log-stderr-prefix` flag)

Patch the `rook-ceph-mon-b` Deployment to run a sleep instead of the `ceph mon` command:

```
kubectl -n rook-ceph patch deployment rook-ceph-mon-b -p '{"spec": {"template": {"spec": {"containers": [{"name": "mon", "command": ["sleep", "infinity"], "args": []}]}}}}'
```

Connect to the pod of a healthy mon and run the following commands.

```
kubectl -n rook-ceph exec -it <mon-pod> bash

```

```
# set a few simple variables
cluster_namespace=rook-ceph
good_mon_id=b
monmap_path=/tmp/monmap

# extract the monmap to a file, by pasting the ceph mon command
# from the good mon deployment and adding the
# `--extract-monmap=${monmap_path}` flag
ceph-mon \
    --fsid=41a537f2-f282-428e-989f-a9e07be32e47 \
    --keyring=/etc/ceph/keyring-store/keyring \
    --log-to-stderr=true \
    --err-to-stderr=true \
    --mon-cluster-log-to-stderr=true \
    --log-stderr-prefix=debug \
    --default-log-to-file=false \
    --default-mon-cluster-log-to-file=false \
    --mon-host=$(ROOK_CEPH_MON_HOST) \
    --mon-initial-members=$(ROOK_CEPH_MON_INITIAL_MEMBERS) \
    --id=b \
    --setuser=ceph \
    --setgroup=ceph \
    --foreground \
    --public-addr=10.100.13.242 \
    --setuser-match-path=/var/lib/ceph/mon/ceph-b/store.db \
    --public-bind-addr=$(ROOK_POD_IP) \
    --extract-monmap=${monmap_path}

# review the contents of the monmap
monmaptool --print /tmp/monmap

# remove the bad mon(s) from the monmap
monmaptool ${monmap_path} --rm <bad_mon>

# in this example we remove mon0 and mon2:
monmaptool ${monmap_path} --rm a
monmaptool ${monmap_path} --rm c

# inject the modified monmap into the good mon, by pasting
# the ceph mon command and adding the
# `--inject-monmap=${monmap_path}` flag, like this
ceph-mon \
    --fsid=41a537f2-f282-428e-989f-a9e07be32e47 \
    --keyring=/etc/ceph/keyring-store/keyring \
    --log-to-stderr=true \
    --err-to-stderr=true \
    --mon-cluster-log-to-stderr=true \
    --log-stderr-prefix=debug \
    --default-log-to-file=false \
    --default-mon-cluster-log-to-file=false \
    --mon-host=$(ROOK_CEPH_MON_HOST) \
    --mon-initial-members=$(ROOK_CEPH_MON_INITIAL_MEMBERS) \
    --id=b \
```

```
    --setuser=ceph \
    --setgroup=ceph \
    --foreground \
    --public-addr=10.100.13.242 \
    --setuser-match-path=/var/lib/ceph/mon/ceph-b/store.db \
    --public-bind-addr=$(ROOK_POD_IP) \
    --inject-monmap=${monmap_path}
```

Exit the shell to continue.

Edit the Rook configmaps

Edit the configmap that the operator uses to track the mons.

```
kubectl -n rook-ceph edit configmap rook-ceph-mon-endpoints
```

In the `data` element you will see three mons such as the following (or more depending on your `moncount`):

```
data: a=10.100.35.200:6789;b=10.100.13.242:6789;c=10.100.35.12:6789
```

Delete the bad mons from the list, for example to end up with a single good mon:

```
data: b=10.100.13.242:6789
```

Save the file and exit.

Now we need to adapt a Secret which is used for the mons and other components. The following `kubectl patch` command is an easy way to do that. In the end it patches the `rook-ceph-config` secret and updates the two key/value pairs `mon_host` and `mon_initial_members`.

```
mon_host=$(kubectl -n rook-ceph get svc rook-ceph-mon-b -o jsonpath='{.spec.clusterIP}')
kubectl -n rook-ceph patch secret rook-ceph-config -p '{"stringData": {"mon_host": "
[v2:'"${mon_host}"':3300,v1:'"${mon_host}"':6789]", "mon_initial_members": "'"${good_mon_id}"'"}}'
```

> **NOTE**: If you are using `hostNetwork: true`, you need to replace the `mon_host` var with the node IP the mon is pinned to (`nodeSelector`). This is because there is no `rook-ceph-mon-*` service created in that "mode".

Restart the mon

You will need to "restart" the good mon pod with the original `ceph-mon` command to pick up the changes. For this run `kubectl replace` on the backup of the mon deployment yaml:

```
kubectl replace --force -f rook-ceph-mon-b-deployment.yaml
```

> **NOTE**: Option `--force` will delete the deployment and create a new one

Start the rook toolbox and verify the status of the cluster.

```
ceph -s
```

The status should show one mon in quorum. If the status looks good, your cluster should be healthy again.

Restart the operator

Start the rook operator again to resume monitoring the health of the cluster.

```
# create the operator. it is safe to ignore the errors that a number of resources already exist.
kubectl -n rook-ceph scale deployment rook-ceph-operator --replicas=1
```

The operator will automatically add more mons to increase the quorum size again, depending on the `mon.count` .

Adopt an existing Rook Ceph cluster into a new Kubernetes cluster

Situations this section can help resolve

1. The Kubernetes environment underlying a running Rook Ceph cluster failed catastrophically, requiring a new Kubernetes environment in which the user wishes to recover the previous Rook Ceph cluster.
2. The user wishes to migrate their existing Rook Ceph cluster to a new Kubernetes environment, and downtime can be tolerated.

Prerequisites

1. A working Kubernetes cluster to which we will migrate the previous Rook Ceph cluster.
2. At least one Ceph mon db is in quorum, and sufficient number of Ceph OSD is `up` and `in` before disaster.
3. The previous Rook Ceph cluster is not running.

Overview for Steps below

1. Start a new and clean Rook Ceph cluster, with old `CephCluster` `CephBlockPool` `CephFilesystem` `CephNFS` `CephObjectStore`.
2. Shut the new cluster down when it has been created successfully.
3. Replace ceph-mon data with that of the old cluster.
4. Replace `fsid` in `secrets/rook-ceph-mon` with that of the old one.
5. Fix monmap in ceph-mon db.
6. Fix ceph mon auth key.
7. Disable auth.
8. Start the new cluster, watch it resurrect.
9. Fix admin auth key, and enable auth.
10. Restart cluster for the final time.

Steps

Assuming `dataHostPathData` is `/var/lib/rook`, and the `CephCluster` trying to adopt is named `rook-ceph`.

1. Make sure the old Kubernetes cluster is completely torn down and the new Kubernetes cluster is up and running without Rook Ceph.
2. Backup `/var/lib/rook` in all the Rook Ceph nodes to a different directory. Backups will be used later.
3. Pick a `/var/lib/rook/rook-ceph/rook-ceph.config` from any previous Rook Ceph node and save the old cluster `fsid` from its content.
4. Remove `/var/lib/rook` from all the Rook Ceph nodes.
5. Add identical `CephCluster` descriptor to the new Kubernetes cluster, especially identical `spec.storage.config` and `spec.storage.nodes`, except `mon.count`, which should be set to `1`.

6. Add identical `CephFilesystem` `CephBlockPool` `CephNFS` `CephObjectStore` descriptors (if any) to the new Kubernetes cluster.
7. Install Rook Ceph in the new Kubernetes cluster.
8. Watch the operator logs with `kubectl -n rook-ceph logs -f rook-ceph-operator-xxxxxxx`, and wait until the orchestration has settled.
9. **STATE**: Now the cluster will have `rook-ceph-mon-a`, `rook-ceph-mgr-a`, and all the auxiliary pods up and running, and zero (hopefully) `rook-ceph-osd-ID-xxxxxx` running. `ceph -s` output should report 1 mon, 1 mgr running, and all of the OSDs down, all PGs are in `unknown` state. Rook should not start any OSD daemon since all devices belongs to the old cluster (which have a different `fsid`).
10. Run `kubectl -n rook-ceph exec -it rook-ceph-mon-a-xxxxxxxx bash` to enter the `rook-ceph-mon-a` pod,

```
mon-a# cat /etc/ceph/keyring-store/keyring  # save this keyring content for later use
mon-a# exit
```

11. Stop the Rook operator by running `kubectl -n rook-ceph edit deploy/rook-ceph-operator` and set `replicas` to `0`.
12. Stop cluster daemons by running `kubectl -n rook-ceph delete deploy/X` where X is every deployment in namespace `rook-ceph`, except `rook-ceph-operator` and `rook-ceph-tools`.
13. Save the `rook-ceph-mon-a` address with `kubectl -n rook-ceph get cm/rook-ceph-mon-endpoints -o yaml` in the new Kubernetes cluster for later use.
14. SSH to the host where `rook-ceph-mon-a` in the new Kubernetes cluster resides.
    i. Remove `/var/lib/rook/mon-a`
    ii. Pick a healthy `rook-ceph-mon-ID` directory (`/var/lib/rook/mon-ID`) in the previous backup, copy to `/var/lib/rook/mon-a`. `ID` is any healthy mon node ID of the old cluster.
    iii. Replace `/var/lib/rook/mon-a/keyring` with the saved keyring, preserving only the `[mon.]` section, remove `[client.admin]` section.
    iv. Run `docker run -it --rm -v /var/lib/rook:/var/lib/rook ceph/ceph:v14.2.1-20190430 bash`. The Docker image tag should match the Ceph version used in the Rook cluster. The `/etc/ceph/ceph.conf` file needs to exist for `ceph-mon` to work.

```
container# touch /etc/ceph/ceph.conf
container# cd /var/lib/rook
container# ceph-mon --extract-monmap monmap --mon-data ./mon-a/data  # Extract monmap from old ceph-mon db and save as monmap
container# monmaptool --print monmap  # Print the monmap content, which reflects the old cluster ceph-mon configuration.
container# monmaptool --rm a monmap  # Delete `a` from monmap.
container# monmaptool --rm b monmap  # Repeat, and delete `b` from monmap.
container# monmaptool --rm c monmap  # Repeat this pattern until all the old ceph-mons are removed
container# monmaptool --rm d monmap
container# monmaptool --rm e monmap
```

```
container# monmaptool --addv a [v2:10.77.2.216:3300,v1:10.77.2.216:6789] monmap   # Replace it with the rook-ceph-mon-a address you got from previous command.
container# ceph-mon --inject-monmap monmap --mon-data ./mon-a/data  # Replace monmap in ceph-mon db with our modified version.
container# rm monmap
container# exit
```

15. Tell Rook to run as old cluster by running `kubectl -n rook-ceph edit secret/rook-ceph-mon` and changing `fsid` to the original `fsid`. Note that the `fsid` is base64 encoded and must not contain a trailing carriage return. For example:

```
echo -n a811f99a-d865-46b7-8f2c-f94c064e4356 | base64  # Replace with the fsid from your old cluster.
```

16. Disable authentication by running `kubectl -n rook-ceph edit cm/rook-config-override` and adding content below:

```
data:
config: |
    [global]
    auth cluster required = none
    auth service required = none
    auth client required = none
    auth supported = none
```

17. Bring the Rook Ceph operator back online by running `kubectl -n rook-ceph edit deploy/rook-ceph-operator` and set `replicas` to `1`.
18. Watch the operator logs with `kubectl -n rook-ceph logs -f rook-ceph-operator-xxxxxxx`, and wait until the orchestration has settled.
19. **STATE**: Now the new cluster should be up and running with authentication disabled. `ceph -s` should report 1 mon & 1 mgr & all of the OSDs up and running, and all PGs in either `active` or `degraded` state.
20. Run `kubectl -n rook-ceph exec -it rook-ceph-tools-XXXXXXX bash` to enter tools pod:

```
tools# vi key
[paste keyring content saved before, preserving only `[client admin]` section]
tools# ceph auth import -i key
tools# rm key
```

21. Re-enable authentication by running `kubectl -n rook-ceph edit cm/rook-config-override` and removing auth configuration added in previous steps.
22. Stop the Rook operator by running `kubectl -n rook-ceph edit deploy/rook-ceph-operator` and set `replicas` to `0`.
23. Shut down entire new cluster by running `kubectl -n rook-ceph delete deploy/X` where X is every deployment in namespace `rook-ceph`, except `rook-ceph-operator` and `rook-ceph-tools`, again. This time OSD daemons are present and should be removed too.
24. Bring the Rook Ceph operator back online by running `kubectl -n rook-ceph edit deploy/rook-

`ceph-operator` and set `replicas` to `1`.

25. Watch the operator logs with `kubectl -n rook-ceph logs -f rook-ceph-operator-xxxxxxx`, and wait until the orchestration has settled.
26. **STATE**: Now the new cluster should be up and running with authentication enabled. `ceph -s` output should not change much comparing to previous steps.

Tectonic Configuration

Here is a running guide on how to implement Rook on Tectonic. A complete guide on how to install Tectonic is out of the scope of the Rook project. More info can be found on the Tectonic website

Prerequisites

- An installed tectonic-installer. These steps are described on the Tectonic website
- A running matchbox node which will do the provisioning (Matchbox is only required if you are running Tectonic on Bare metal)
- You can run through all steps of the GUI installer, but in the last step, choose `Boot manually`. This way we can make the necessary changes first.

Edit the kubelet.service file

We need to make a few adaptions to the Kubelet systemd service file generated by the Tectonic-installer.

First change to the directory in which you untarred the tectonic installer and find your newly generated cluster configuration files.

```
cd ~/tectonic/tectonic-installer/LINUX-OR-DARWIN/clusters
```

Open the file `modules/ignition/resources/services/kubelet.service` in your favorite editor and after the last line containing `ExecStartPre=...`, paste the following extra lines:

```
ExecStartPre=/bin/mkdir -p /var/lib/kubelet/volumeplugins
ExecStartPre=/bin/mkdir -p /var/lib/rook
```

And after the `ExecStart=/usr/lib/coreos/kubelet-wrapper \` line, insert the following flag for the kubelet-wrapper to point to a path reachable outside of the Kubelet rkt container:

```
--volume-plugin-dir=/var/lib/kubelet/volumeplugins \
```

Save and close the file.

Boot your Tectonic cluster

All the preparations are ready for Tectonic to boot now. We will use `terraform` to start the cluster. Visit the official Tectonic manual boot page for the commands to use.

Remark: The Tectonic installer contains the correct terraform binary out of the box. This terraform binary can be found in following directory `~/tectonic/tectonic-installer/linux` .

Start Rook

After the Tectonic Installer ran and the Kubernetes cluster is started and ready, you can follow the Rook installation guide. If you want to specify which disks Rook uses, follow the instructions in creating Rook clusters

Contributing

Thank you for your time and effort to help us improve Rook! Here are a few steps to get started. If you have any questions, don't hesitate to reach out to us on our Slack dev channel.

Prerequisites

1. GO 1.13 or greater installed
2. Git client installed
3. Github account

Initial Setup

Create a Fork

From your browser navigate to http://github.com/rook/rook and click the "Fork" button.

Clone Your Fork

Open a console window and do the following;

```
# Create the rook repo path
mkdir -p $GOPATH/src/github.com/rook

# Navigate to the local repo path and clone your fork
cd $GOPATH/src/github.com/rook

# Clone your fork, where <user> is your GitHub account name
git clone https://github.com/<user>/rook.git
cd rook
```

Build

```
# build all rook storage providers
make

# build a single storage provider, where the IMAGES can be a subdirectory of the "images" folder:
# "cassandra", "ceph", "cockroachdb", "edgefs", or "nfs"
make IMAGES="cassandra" build

# multiple storage providers can also be built
make IMAGES="cassandra ceph" build
```

Development Settings

To provide consistent whitespace and other formatting in your `go` and other source files (e.g., Markdown), it is recommended you apply the following settings in your IDE:

- Format with the `goreturns` tool
- Trim trailing whitespace
- Markdown Table of Contents is correctly updated automatically

For example, in VS Code this translates to the following settings:

```
{
    "editor.formatOnSave": true,
    "go.buildOnSave": "package",
    "go.formatTool": "goreturns",
    "files.trimTrailingWhitespace": true,
    "files.insertFinalNewline": true,
    "files.trimFinalNewlines": true,
    "markdown.extension.toc.unorderedList.marker": "*",
    "markdown.extension.toc.githubCompatibility": true,
    "markdown.extension.toc.levels": "2..2"
}
```

In addition to that it is recommended to install the following extensions:

- Markdown All in One by Yu Zhang - Visual Studio Marketplace

Add Upstream Remote

First you will need to add the upstream remote to your local git:

```
# Add 'upstream' to the list of remotes
git remote add upstream https://github.com/rook/rook.git

# Verify the remote was added
git remote -v
```

Now you should have at least `origin` and `upstream` remotes. You can also add other remotes to collaborate with other contributors.

Layout

A source code layout is shown below, annotated with comments about the use of each important directory:

```
rook
├── build                         # build makefiles and logic to build, publish and release all Rook artifacts
```

```
├── cluster
│   ├── charts                    # Helm charts
│   │   └── rook-ceph
│   └── examples                  # Sample yaml files for Rook cluster
│
├── cmd                           # Binaries with main entrypoint
│   ├── rook                      # Main command entry points for operators and daemons
│   └── rookflex                  # Main command entry points for Rook flexvolume driver
│
├── design                        # Design documents for the various components of the Rook project
├── Documentation                 # Rook project Documentation
├── images                        # Dockerfiles to build images for all supported storage providers
│
├── pkg
│   ├── apis
│   │   ├── ceph.rook.io          # ceph specific specs for cluster, file, object
│   │   │   ├── v1
│   │   ├── cockroachdb.rook.io   # cockroachdb specific specs
│   │   │   └── v1alpha1
│   │   ├── nfs.rook.io           # nfs server specific specs
│   │   │   └── v1alpha1
│   │   └── rook.io               # rook.io API group of common types
│   │       └── v1alpha2
│   ├── client                    # auto-generated strongly typed client code to access Rook APIs
│   ├── clusterd
│   ├── daemon                    # daemons for each storage provider
│   │   ├── ceph
│   │   └── discover
│   ├── operator                  # all orchestration logic and custom controllers for each storage provider
│   │   ├── ceph
│   │   ├── cockroachdb
│   │   ├── discover
│   │   ├── k8sutil
│   │   ├── nfs
│   │   └── test
│   ├── test
│   ├── util
│   └── version
└── tests                         # integration tests
    ├── framework                 # the Rook testing framework
    │   ├── clients               # test clients used to consume Rook resources during integration tests
    │   ├── installer             # installs Rook and its supported storage providers into integration tests
environments
    │   └── utils
    ├── integration               # all test cases that will be invoked during integration testing
    ├── longhaul                  # longhaul tests
    ├── pipeline                  # Jenkins pipeline
    └── scripts                   # scripts for setting up integration and manual testing environments
```

Development

To add a feature or to make a bug fix, you will need to create a branch in your fork

and then submit a pull request (PR) from the branch.

Design Document

For new features of significant scope and complexity, a design document is recommended before work begins on the implementation. So create a design document if:

- Adding a new storage provider
- Adding a new CRD
- Adding a significant feature to an existing storage provider. If the design is simple enough to describe in a github issue, you likely don't need a full design doc.

For smaller, straightforward features and bug fixes, there is no need for a design document. Authoring a design document for big features has many advantages:

- Helps flesh out the approach by forcing the author to think critically about the feature and can identify potential issues early on
- Gets agreement amongst the community before code is written that could be wasted effort in the wrong direction
- Serves as an artifact of the architecture that is easier to read for visitors to the project than just the code by itself

Note that writing code to prototype the feature while working on the design may be very useful to help flesh out the approach.

A design document should be written as a markdown file in the design folder. You can follow the process outlined in the design template. You will see many examples of previous design documents in that folder. Submit a pull request for the design to be discussed and approved by the community before being merged into master, just like any other change to the repository.

An issue should be opened to track the work of authoring and completing the design document. This issue is in addition to the issue that is tracking the implementation of the feature. The design label should be assigned to the issue to denote it as such.

Create a Branch

From a console, create a new branch based on your fork and start working on it:

```
# Ensure all your remotes are up to date with the latest
git fetch --all

# Create a new branch that is based off upstream master.  Give it a simple, but descriptive name.
# Generally it will be two to three words separated by dashes and without numbers.
git checkout -b feature-name upstream/master
```

Now you are ready to make the changes and commit to your branch.

Updating Your Fork

During the development lifecycle, you will need to keep up-to-date with the latest upstream master. As others on the team push changes, you will need to `rebase` your commits on top of the latest. This avoids unnecessary merge commits and keeps the commit history clean.

Whenever you need to update your local repository, you never want to merge. You **always** will rebase. Otherwise you will end up with merge commits in the git history. If you have any modified files, you will first have to stash them (`git stash save -u "<some description>"`).

```
1. git fetch --all
2. git rebase upstream/master
```

Rebasing is a very powerful feature of Git. You need to understand how it works or else you will risk losing your work. Read about it in the Git documentation, it will be well worth it. In a nutshell, rebasing does the following:

- “Unwinds” your local commits. Your local commits are removed temporarily from the history.
- The latest changes from upstream are added to the history
- Your local commits are re-applied one by one
- If there are merge conflicts, you will be prompted to fix them before continuing. Read the output closely. It will tell you how to complete the rebase.
- When done rebasing, you will see all of your commits in the history.

Submitting a Pull Request

Once you have implemented the feature or bug fix in your branch, you will open a PR to the upstream rook repo. Before opening the PR ensure you have added unit tests, are passing the integration tests, cleaned your commit history, and have rebased on the latest upstream.

In order to open a pull request (PR) it is required to be up to date with the latest changes upstream. If other commits are pushed upstream before your PR is merged, you will also need to rebase again before it will be merged.

Regression Testing

All pull requests must pass the unit and integration tests before they can be merged. These tests automatically run as a part of the build process. The results of these tests along with code reviews and other criteria determine whether your request will be accepted into the `rook/rook` repo. It is prudent to run all tests locally on your development box prior to submitting a pull request to the `rook/rook` repo.

Unit Tests

From the root of your local Rook repo execute the following to run all of the unit tests:

```
make test
```

Unit tests for individual packages can be run with the standard `go test` command. Before you open a PR, confirm that you have sufficient code coverage on the packages that you changed. View the `coverage.html` in a browser to inspect your new code.

```
go test -coverprofile=coverage.out
go tool cover -html=coverage.out -o coverage.html
```

Running the Integration Tests

For instructions on how to execute the end to end smoke test suite, follow the test instructions.

Commit structure

Rook maintainers value clear, lengthy and explanatory commit messages. So by default each of your commits must:

- be prefixed by the component it's affecting, if Ceph, then the title of the commit message should be `ceph: my commit title`. If not the commit-lint bot will complain.
- contain a commit message which explains the original issue and how it was fixed if a bug. If a feature it is a full description of the new functionality.
- refer to the issue it's closing, this is mandatory when fixing a bug
- have a sign-off, this is achieved by adding `-s` when committing so in practice run `git commit -s`. If not the DCO bot will complain. If you forgot to add the sign-off you can also amend a previous commit with the sign-off by running `git commit --amend -s`. If you've pushed your changes to Github already you'll need to force push your branch with `git push -f`.

Here is an example of an acceptable commit message:

```
component: commit title

This is the commit message, here I'm explaining, what the bug was along with its root cause.
Then I'm explaining how I fixed it.

Closes: https://github.com/rook/rook/issues/<NUMBER>
Signed-off-by: First Name Last Name <email address>
```

The `component` **MUST** be one of the following:

- bot
- build
- cassandra
- ceph
- ci
- cockroachdb
- core
- docs
- edgefs
- nfs
- test
- yugabytedb

Note: sometimes you will feel like there is not so much to say, for instance if you are fixing a typo in a text. In that case, it is acceptable to shorten the commit message. Also, you don't always need to close an issue, again for a very small fix.

You can read more about conventional commits.

Commit History

To prepare your branch to open a PR, you will need to have the minimal number of logical commits so we can maintain a clean commit history. Most commonly a PR will include a single commit where all changes are squashed, although sometimes there will be multiple logical commits.

```
# Inspect your commit history to determine if you need to squash commits
git log

# Rebase the commits and edit, squash, or even reorder them as you determine will keep the history clean.
# In this example, the last 5 commits will be opened in the git rebase tool.
git rebase -i HEAD~5
```

Once your commit history is clean, ensure you have based on the latest upstream before you open the PR.

Submitting

Go to the Rook github to open the PR. If you have pushed recently, you should see an obvious link to open the PR. If you have not pushed recently, go to the Pull Request tab and select your fork and branch for the PR.

After the PR is open, you can make changes simply by pushing new commits. Your PR will track the changes in your fork and update automatically.

Never open a pull request against a released branch (e.g. release-1.2) unless the content you are editing is gone from master and only exists in the released branch. By default, you should always open a pull request against master.

Backport a Fix to a Release Branch

The flow for getting a fix into a release branch is:

1. Open a PR to merge the changes to master following the process outlined above.
2. Add the backport label to that PR such as backport-release-1.1
3. After your PR is merged to master, the mergify bot will automatically open a PR with your commits backported to the release branch
4. If there are any conflicts you will need to resolve them by pulling the branch, resolving the conflicts and force push back the branch
5. After the CI is green, the bot will automatically merge the backport PR.

Debugging operators locally

Operators are meant to be run inside a Kubernetes cluster. However, this makes it harder to use debugging tools and slows down the developer cycle of edit-build-test since testing requires to build a container image, push to the cluster, restart the pods, get logs, etc.

A common operator developer practice is to run the operator locally on the developer machine in order to leverage the developer tools and comfort.

In order to support this external operator mode, rook detects if the operator is running outside of the cluster (using standard cluster env) and changes the behavior as follows:

- Connecting to Kubernetes API will load the config from the user `~/.kube/config`.
- Instead of the default `CommandExecutor` this mode uses a `TranslateCommandExecutor` that executes every command issued by the operator to run as a Kubernetes job inside the cluster, so that any tools that the operator needs from its image can be called. For example, in cockroachdb

Building locally

Building a single rook binary for all operators:

```
1. make GO_STATIC_PACKAGES=github.com/rook/rook/cmd/rook go.build
```

Note: the binary output location is `_output/bin/linux_amd64/rook` on linux, and `_output/bin/darwin_amd64/rook` on mac.

Running locally

The command-line flag: `--operator-image <image>` should be used to allow running outside of a pod since some operators read the image from the pod. This is a pattern where the operator pod is based on the image of the actual storage provider image (currently

used by ceph, edgefs, cockroachdb). The image url should be passed manually (for now) to match the operator's Dockerfile `FROM` statement.

The next sections describe the supported operators and their notes.

CockroachDB

```
1. _output/bin/darwin_amd64/rook cockroachdb operator --operator-image cockroachdb/cockroach:v2.0.2
```

- Set `--operator-image` to the base image of cockroachdb Dockerfile
- The execution of `/cockroach/cockroach init` in initCluster() runs in a kubernetes job to complete the clusterization of its pods.

Multi-Node Test Environment

Using KVM/QEMU and Kubespray

Setup expectation

There are a bunch of pre-requisites to be able to deploy the following environment. Such as:

- A Linux workstation (CentOS or Fedora)
- KVM/QEMU installation
- docker service allowing insecure local repository

For other Linux distribution, there is no guarantee the following will work. However adapting commands (apt/yum/dnf) could just work.

Prerequisites installation

On your host machine, execute `tests/scripts/multi-node/rpm-system-prerequisites.sh` (or do the equivalent for your distribution)

Edit `/etc/docker/daemon.json` to add insecure-registries:

```
{
        "insecure-registries":  ["172.17.8.1:5000"]
}
```

Deploy Kubernetes with Kubespray

Clone it:

```
git clone https://github.com/kubernetes-sigs/kubespray/
cd kubespray
```

Edit `inventory/sample/group_vars/k8s-cluster/k8s-cluster.yml` with:

```
docker_options: "--insecure-registry=172.17.8.1:5000 --insecure-registry={{ kube_service_addresses }} --data-root={{ docker_daemon_graph }} {{ docker_log_opts }}"
```

FYI: `172.17.8.1` is the libvirt bridge IP, so it’s reachable from all your virtual

machines. This means a registry running on the host machine is reachable from the virtual machines running the Kubernetes cluster.

Create Vagrant's variable directory:

```
mkdir vagrant/
```

Put `tests/scripts/multi-node/config.rb` in `vagrant/`. You can adapt it at will. Feel free to adapt `num_instances`.

Deploy!

```
vagrant up --no-provision ; vagrant provision
```

Go grab a coffee:

```
PLAY RECAP *********************************************************************
k8s-01                     : ok=351  changed=111  unreachable=0    failed=0
k8s-02                     : ok=230  changed=65   unreachable=0    failed=0
k8s-03                     : ok=230  changed=65   unreachable=0    failed=0
k8s-04                     : ok=229  changed=65   unreachable=0    failed=0
k8s-05                     : ok=229  changed=65   unreachable=0    failed=0
k8s-06                     : ok=229  changed=65   unreachable=0    failed=0
k8s-07                     : ok=229  changed=65   unreachable=0    failed=0
k8s-08                     : ok=229  changed=65   unreachable=0    failed=0
k8s-09                     : ok=229  changed=65   unreachable=0    failed=0

Friday 12 January 2018  10:25:45 +0100 (0:00:00.017)       0:17:24.413 ********
===============================================================================
download : container_download | Download containers if pull is required or told to always pull (all nodes) - 192.44s
kubernetes/preinstall : Update package management cache (YUM) --------- 178.26s
download : container_download | Download containers if pull is required or told to always pull (all nodes) - 102.24s
docker : ensure docker packages are installed -------------------------- 57.20s
download : container_download | Download containers if pull is required or told to always pull (all nodes) - 52.33s
kubernetes/preinstall : Install packages requirements ------------------ 25.18s
download : container_download | Download containers if pull is required or told to always pull (all nodes) - 23.74s
download : container_download | Download containers if pull is required or told to always pull (all nodes) - 18.90s
download : container_download | Download containers if pull is required or told to always pull (all nodes) - 15.39s
kubernetes/master : Master | wait for the apiserver to be running ------ 12.44s
download : container_download | Download containers if pull is required or told to always pull (all nodes) - 11.83s
download : container_download | Download containers if pull is required or told to always pull (all nodes) - 11.66s
kubernetes/node : install | Copy kubelet from hyperkube container ------ 11.44s
download : container_download | Download containers if pull is required or told to always pull (all nodes) - 11.41s
download : container_download | Download containers if pull is required or told to always pull (all nodes) - 11.00s
```

```
docker : Docker | pause while Docker restarts ------------------------- 10.22s
kubernetes/secrets : Check certs | check if a cert already exists on node --- 6.05s
kubernetes-apps/network_plugin/flannel : Flannel | Wait for flannel subnet.env file presence --- 5.33s
kubernetes/master : Master | wait for kube-scheduler -------------------- 5.30s
kubernetes/master : Copy kubectl from hyperkube container --------------- 4.77s
[leseb@tarox kubespray]$
[leseb@tarox kubespray]$
[leseb@tarox kubespray]$ vagrant ssh k8s-01
Last login: Fri Jan 12 09:22:18 2018 from 192.168.121.1

[vagrant@k8s-01 ~]$ kubectl get nodes
NAME      STATUS    ROLES         AGE       VERSION
k8s-01    Ready     master,node   2m        v1.9.0+coreos.0
k8s-02    Ready     node          2m        v1.9.0+coreos.0
k8s-03    Ready     node          2m        v1.9.0+coreos.0
k8s-04    Ready     node          2m        v1.9.0+coreos.0
k8s-05    Ready     node          2m        v1.9.0+coreos.0
k8s-06    Ready     node          2m        v1.9.0+coreos.0
k8s-07    Ready     node          2m        v1.9.0+coreos.0
k8s-08    Ready     node          2m        v1.9.0+coreos.0
k8s-09    Ready     node          2m        v1.9.0+coreos.0
```

Running the Kubernetes Dashboard UI

kubespray sets up the Dashboard pod by default, but you must authenticate with a bearer token, even for localhost access with kubectl proxy. To allow access, one possible solution is to:

1) Create an admin user by creating admin-user.yaml with these contents (and using kubectl -f create admin-user.yaml):

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kube-system
```

2) Grant that user the ClusterRole authorization by creating and applying admin-user-cluster.role.yaml:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
```

```
    name: admin-user
    namespace: kube-system
```

3) Find the admin-user token in the kube-system namespace:

```
kubectl -n kube-system describe secret $(kubectl -n kube-system get secret | grep admin-user | awk '{print $1}')
```

and you can use that token to log into the UI at http://localhost:8001/ui.

(See https://github.com/kubernetes/dashboard/wiki/Creating-sample-user)

Development workflow on the host

Everything should happen on the host, your development environment will reside on the host machine NOT inside the virtual machines running the Kubernetes cluster.

Now, please refer to https://rook.io/docs/rook/master/development-flow.html to setup your development environment (go, git etc).

At this stage, Rook should be cloned on your host.

From your Rook repository (should be $GOPATH/src/github.com/rook) location execute `bash tests/scripts/multi-node/build-rook.sh` . During its execution, `build-rook.sh` will purge all running Rook pods from the cluster, so that your latest container image can be deployed. Furthermore, **all Ceph data and config will be purged** as well. Ensure that you are done with all existing state on your test cluster before executing `build-rook.sh` as it will clear everything.

Each time you build and deploy with `build-rook.sh` , the virtual machines (k8s-0X) will pull the new container image and run your new Rook code. You can run `bash tests/scripts/multi-node/build-rook.sh` as many times as you want to rebuild your new rook image and redeploy a cluster that is running your new code.

From here, resume your dev, change your code and test it by running `bash tests/scripts/multi-node/build-rook.sh` .

Teardown

Typically, to flush your environment you will run the following from within kubespray’s git repository. This action will be performed on the host:

```
[user@host-machine kubespray]$ vagrant destroy -f
```

Also, if you were using `kubectl` on that host machine, you can resurrect your old configuration by renaming `$HOME/.kube/config.before.rook.$TIMESTAMP` with `$HOME/.kube/config` .

If you were not using `kubectl` , feel free to simply remove `$HOME/.kube/config.rook` .

Using VirtualBox and k8s-vagrant-multi-node

Prerequisites

Be sure to follow the prerequisites here: https://github.com/galexrt/k8s-vagrant-multi-node/tree/master#prerequisites.

Quickstart

To start up the environment just run `./tests/scripts/k8s-vagrant-multi-node.sh up`. This will bring up one master and 2 workers by default.

To change the amount of workers to bring up and their resources, be sure to checkout the galexrt/k8s-vagrant-multi-node project README Variables section. Just set or export the variables as you need on the script, e.g., either `NODE_COUNT=5 ./tests/scripts/k8s-vagrant-multi-node.sh up`, or `export NODE_COUNT=5` and then `./tests/scripts/k8s-vagrant-multi-node.sh up`.

For more information or if you are experiencing issues, please create an issue at GitHub galexrt/k8s-vagrant-multi-node.

Using Vagrant on Linux with libvirt

See https://github.com/noahdesu/kubensis.