Ember CSI plugin Documentation

Release 0.9.1

Gorka Eguileor

Jun 06, 2022

Contents

Welcome to the Ember-CSI documentation!

Ember-CSI is a plugin to provision and use block and file storage in containerized workloads on Kubernetes and OpenShift.

The documentation for the site is organized into the following sections:

CHAPTER 1

Introduction

The Container Storage Interface (CSI) is a standard for provision and use block and file storage systems in containerized workloads on Container Orchestration Systems (COs) like OpenShift.

Using this interface new storage systems can be exposed to COs without needing to change the COs code.

Ember-CSI is an Open Source implementation of the CSI specification supporting storage solutions from multiple vendors by leveraging a library called cinderlib that provides an abstraction layer over the storage drivers.

1.1 Features

Ember-CSI supports CSI versions 0.2, 0.3, 1.0, and 1.1 providing the following features:

- Volume provisioning: file and block types
- Volume cloning
- Volume deletion
- Snapshot creation
- Create volume from a snapshot
- Snapshots deletion
- Listing volumes with pagination
- Listing snapshots with pagination
- Attaching/Detaching volumes
- Multi pod attaching (block mode only)
- Storage capacity reporting
- Node probing

1.2 Limitations

There are 2 types of volumes in OpenShift and Kubernetes, Block and File, and while both are supported by Ember-CSI, behind the scenes all the storage drivers in Ember-CSI are for block storage systems.

To provide File volumes from block storage Ember-CSI connects the volumes to the host, formats and present them to the Orchestrator for the containerized workloads.

Since File type volumes are locally attached block volumes they cannot be shared between containers, so the Shared Access (RWX) Access Mode is not supported.

This limitation does not apply to block volumes, that can be mounted in multiple hosts simultaneously and it's the application the one responsible to orchestrate the proper access to the disk.

1.3 Supported drivers

Ember-CSI includes a good number of storage drivers, but due to limitation on hardware availability only a small number of them have been validated at one point or another. In alphabetical order they are:

- HPE3PARFC
- HPE3PARISCSI
- KaminarioISCSI
- LVMVolume
- PowerMaxFC
- PowerMaxISCSI
- PureFC
- PureISCSI
- QnapISCSI
- RBD
- SolidFire
- SynoISCSI
- XtremIOFC
- XtremIOISCSI

The remaining drivers included in Ember-CSI have not been validated yet:

- ACCESSIscsi
- AS13000
- FJDXFC
- FJDXISCSI
- FlashSystemFC
- FlashSystemISCSI
- GPFS
- GPFSRemote

- HPELeftHandISCSI
- HPMSAFC
- HPMSAISCSI
- HedvigISCSI
- HuaweiFC
- HuaweiISCSI
- IBMStorage
- InStorageMCSFC
- InStorageMCSISCSI
- InfortrendCLIFC
- InfortrendCLIISCSI
- LenovoFC
- LenovoISCSI
- LinstorDrbd
- LinstorIscsi
- MStorageFC
- MStorageISCSI
- MacroSANFC
- MacroSANISCSI
- NetAppCmodeFibreChannel
- NetAppCmodeISCSI
- NexentaISCSI
- PSSeriesISCSI
- Quobyte
- RSD
- SCFC
- SCISCSI
- SPDK
- Sheepdog
- StorPool
- StorwizeSVCFC
- StorwizeSVCISCSI
- Unity
- VNX
- VZStorage
- VxFlexOS

- WindowsISCSI
- WindowsSmbfs
- ZadaraVPSAISCS

CHAPTER 2

Installation

2.1 Requirements

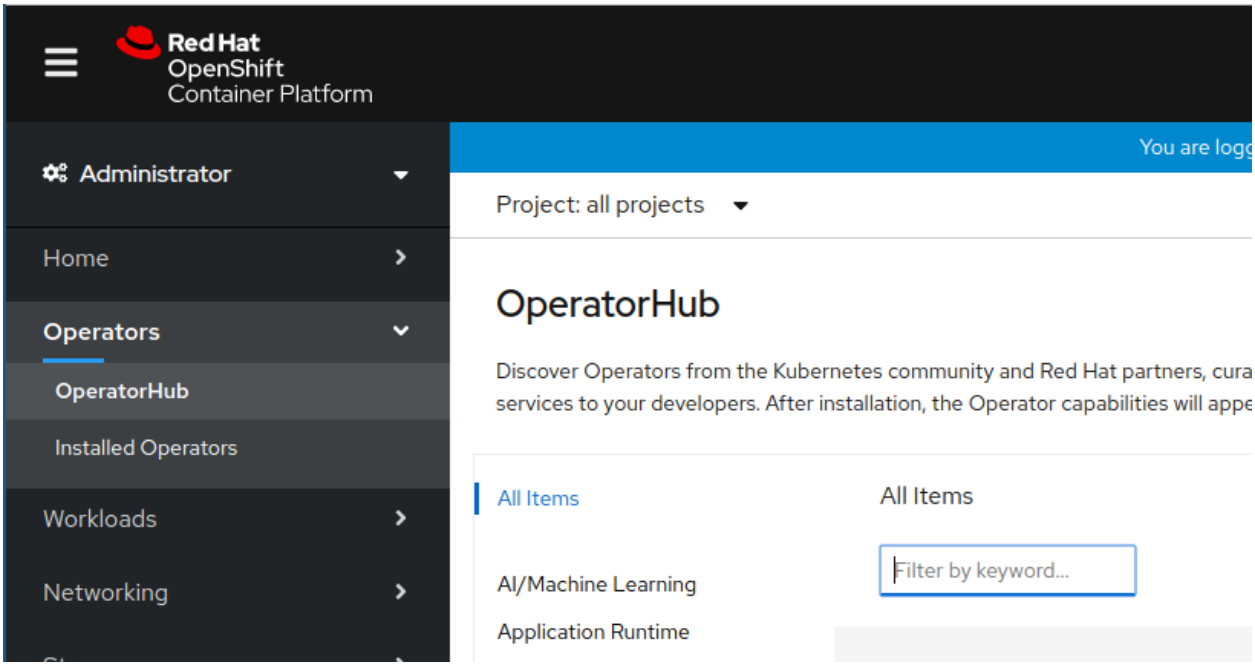
Ember-CSI has the following requirements:

- Operating system: A Red Hat Linux distribution (support for other distributions is possible but not currently provided by the project): - Centos 7 and Centos 8 - RHEL 7 and RHEL 8 - Fedora
- Container Orchestrator: Both Kubernetes and OpenShift are supported: - Openshift: Recommended version 4.6 or newer. Supported from version 4.3 onward. - Kubernetes: Recommended version 1.17 or newer. Supported from version 1.16 onward.
- Storage solution: Access and credentials to a supported storage solution.
- Network: Network connections must be setup appropriately. - Controller nodes: Must have access to the storage management interface. Some drivers require access to the storage data network as well. - Worker nodes: Must have access to the storage data network.
- Services: Depending on the driver and the configuration we may need additional services running on worker nodes and in some drivers controller nodes.
  - iSCSI: For iSCSI drivers the iSCSI initiator, *iscsid* provided by the *iscsi-initiator-utils* package must be configured and running on the host.
  - Multipathing: When selecting multipathing on iSCSI and FC drivers we'll need to have *multipathd*, provided by the *device-mapper-multipath* package, configured and running on the host.

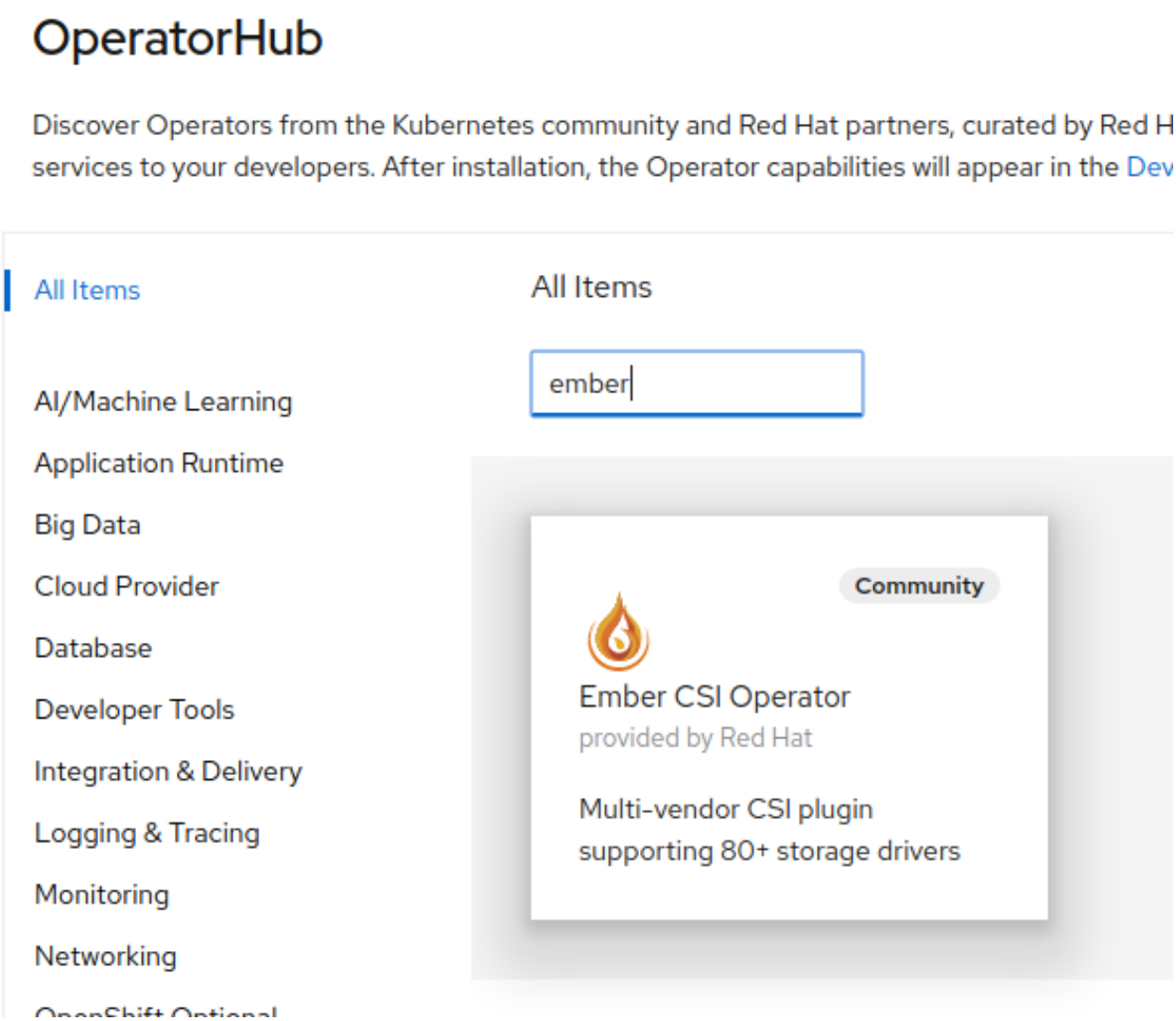
2.2 OpenShift

To eliminate configuration and deployment complexities and the errors that come with them, the recommended mechanism to deploy Ember-CSI is using its operator, which makes installing Ember-CSI a simple and intuitive process.

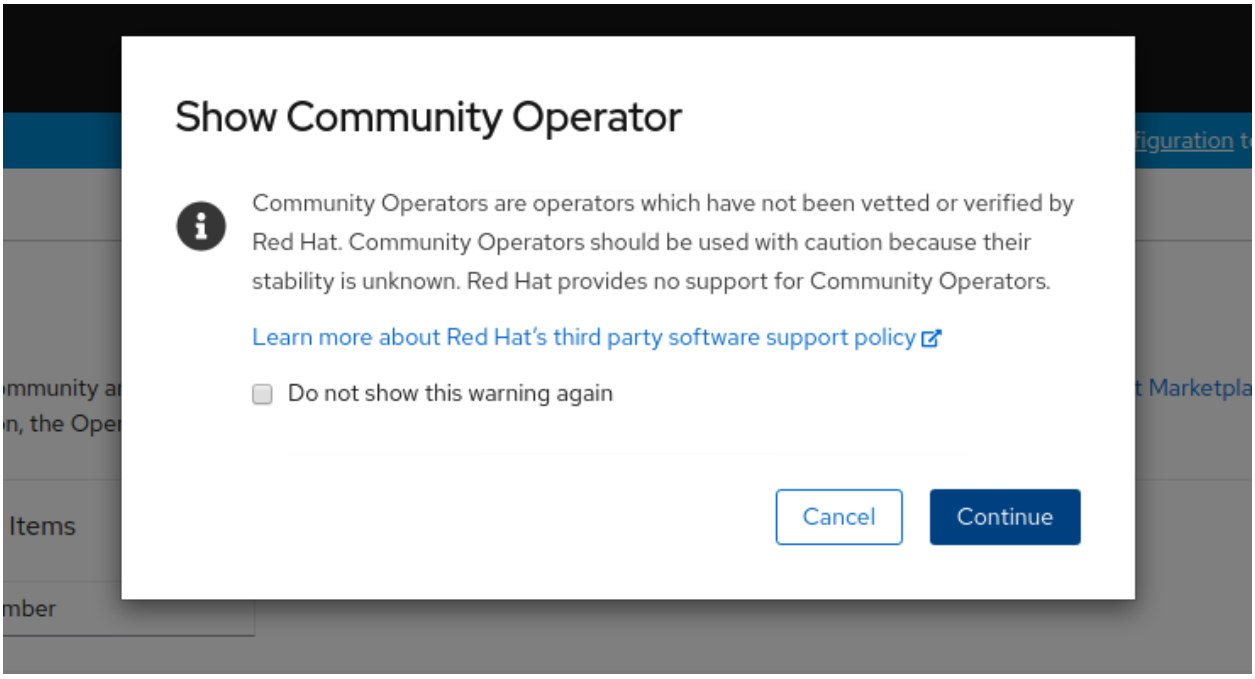
After logging in the OpenShift console as an administrator we go to the OperatorHub:



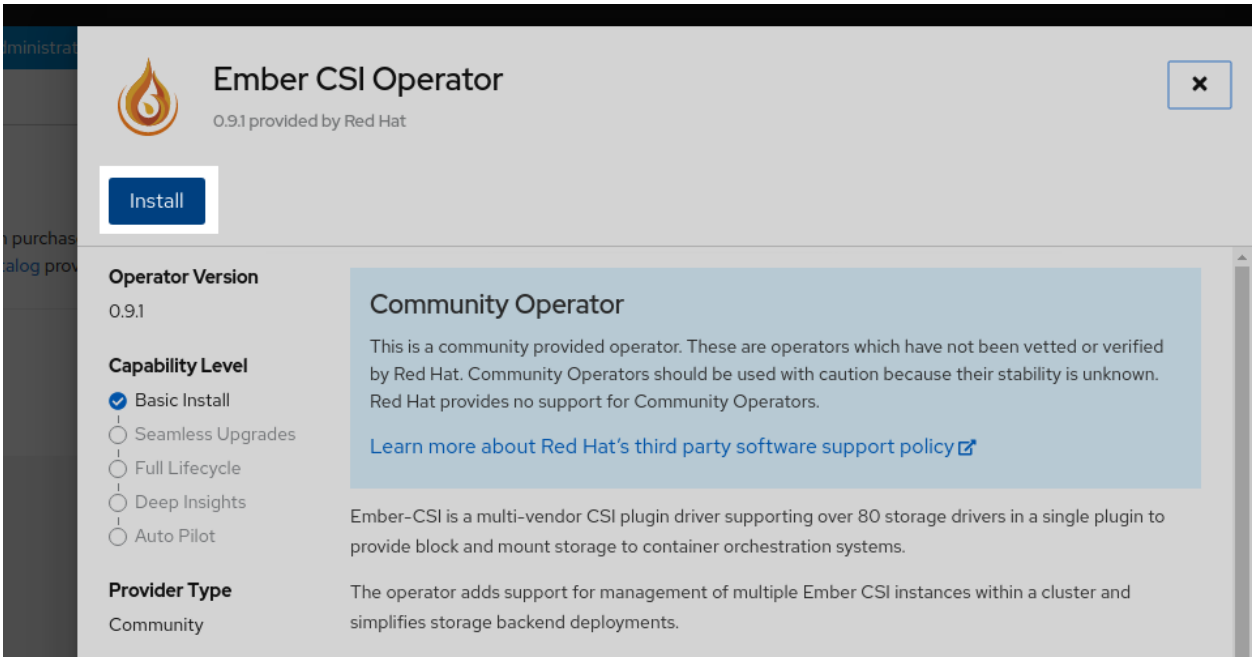
Then we search for the Ember-CSI operator and click on it:



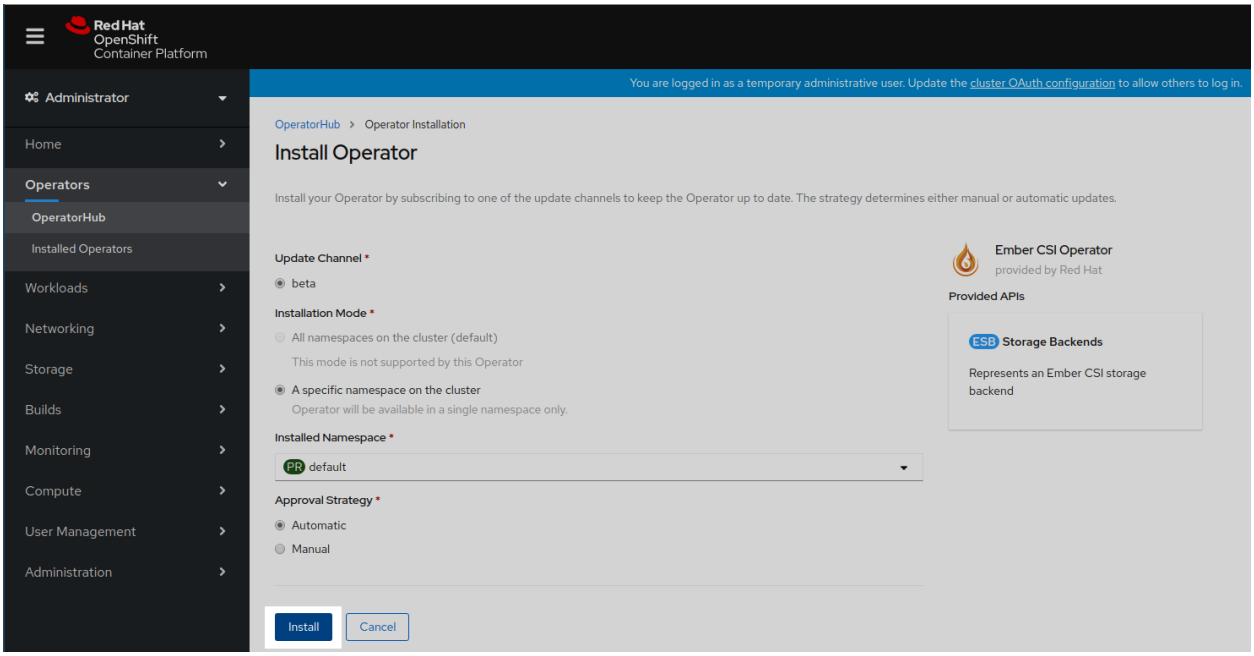
If we are installing the Community Operator we'll be required to confirm that we understand the implications. We click `Continue`:



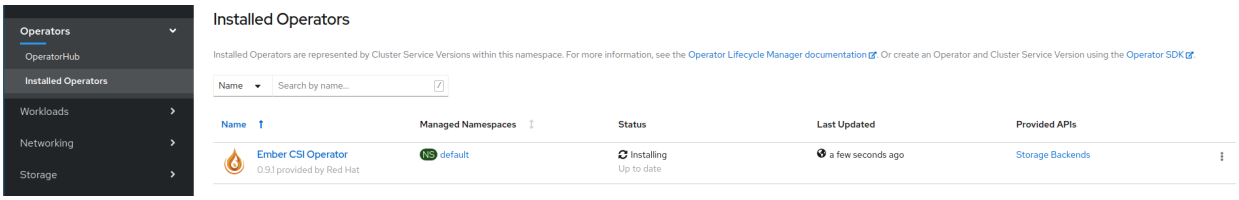
And we are presented with the Ember-CSI Operator page, where we click `Install`:



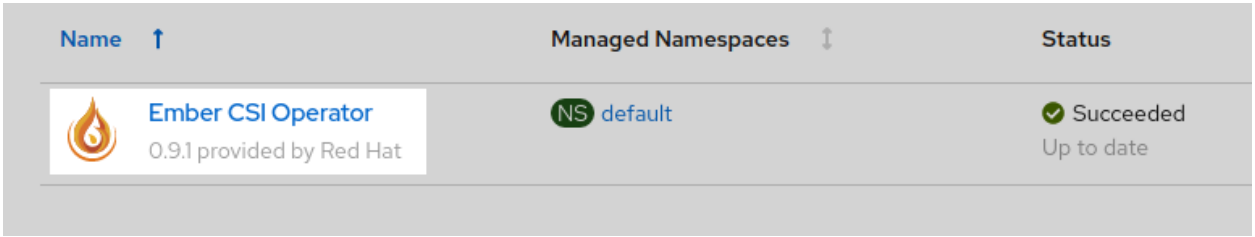
And then `Install` again:



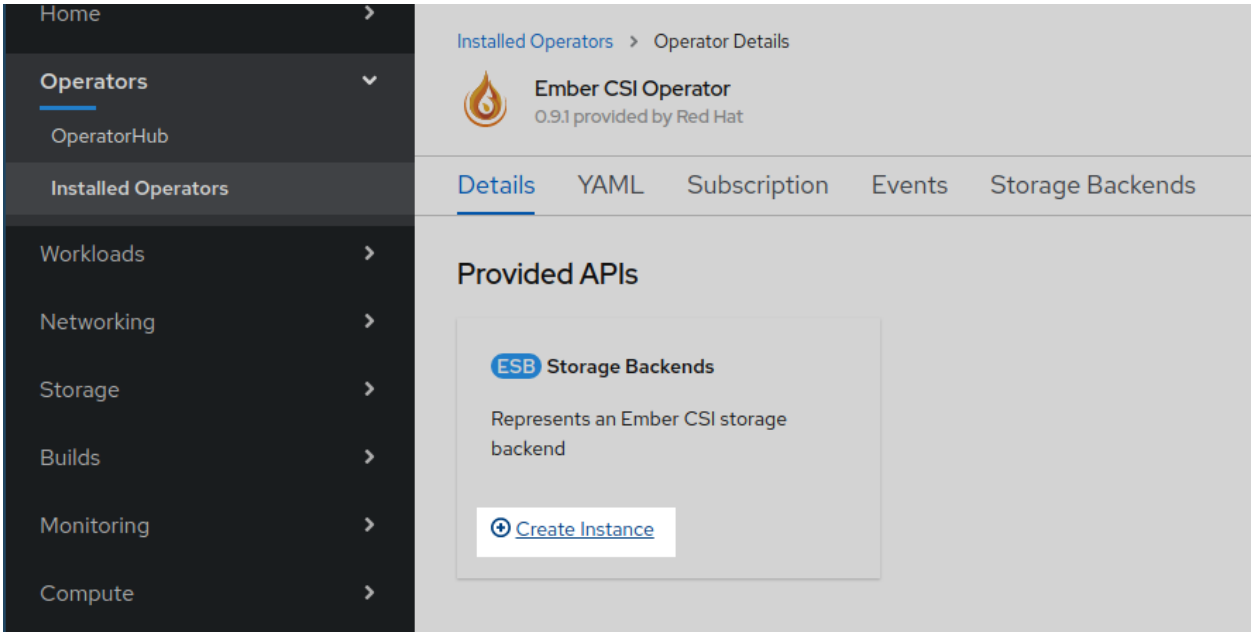
This will trigger the download and execution of the operator container image. It will take a couple of seconds, and in the meantime we'll see that the installation is in progress and maybe a couple of weird entries saying at the beginning:



Once the operator reaches the `Succeeded` status we click on it:

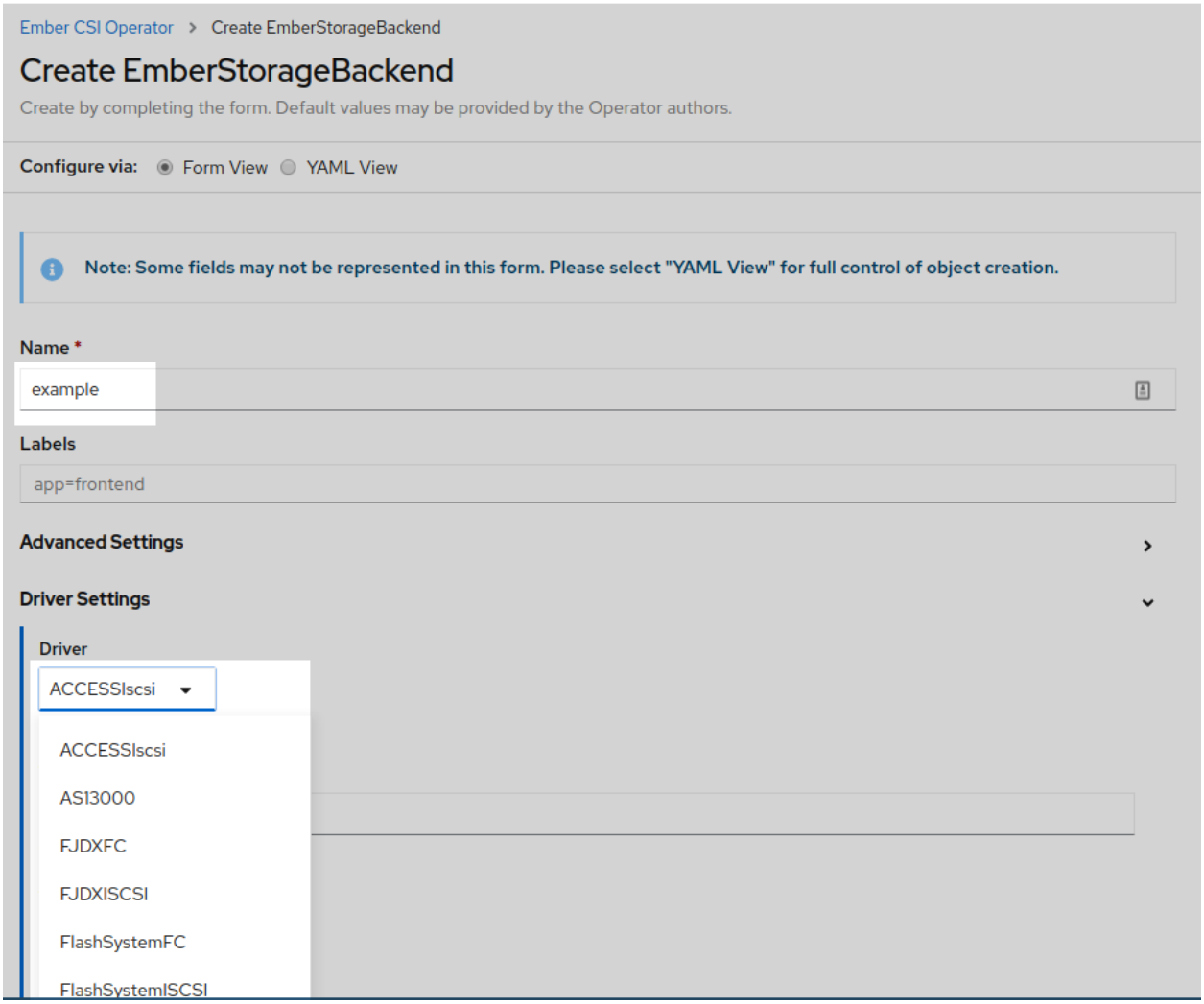


Inside the Ember-CSI operator page we create a new `Storage Backend` instance:

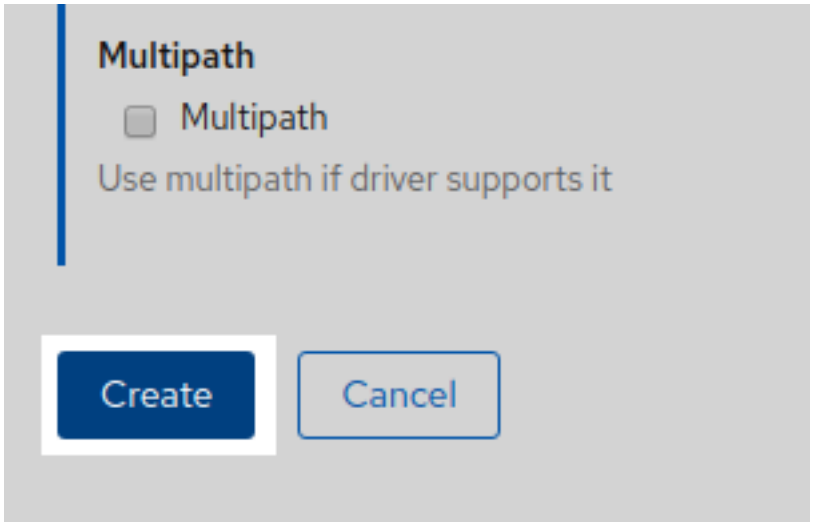


Backends can be configured using YAML, but this is a cumbersome process, so it's usually only used on automated processes such as CI systems, and the Ember-CSI team recommends using the form interface when doing things manually, which is the default on OpenShift 4.5.

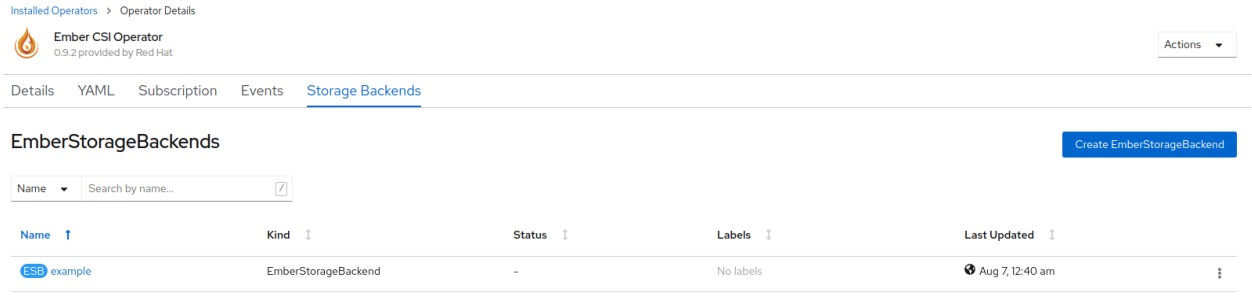
In the form we should change the *name* field from *default* to a unique and meaningful name to identify this backend. Then go to the *Driver* dropdown and select the name of our storage backend. After selecting the appropriate driver the relevant configuration options for the selected driver will be displayed.



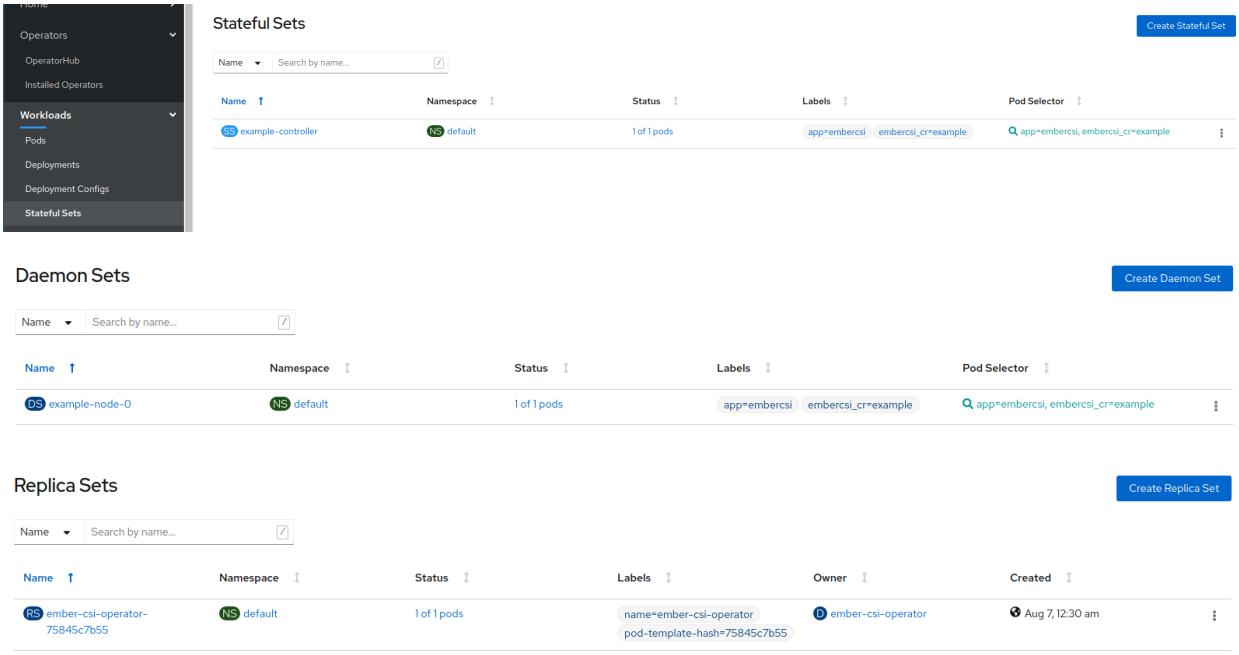
After setting the configuration options we click *Create* at the botom of the page:



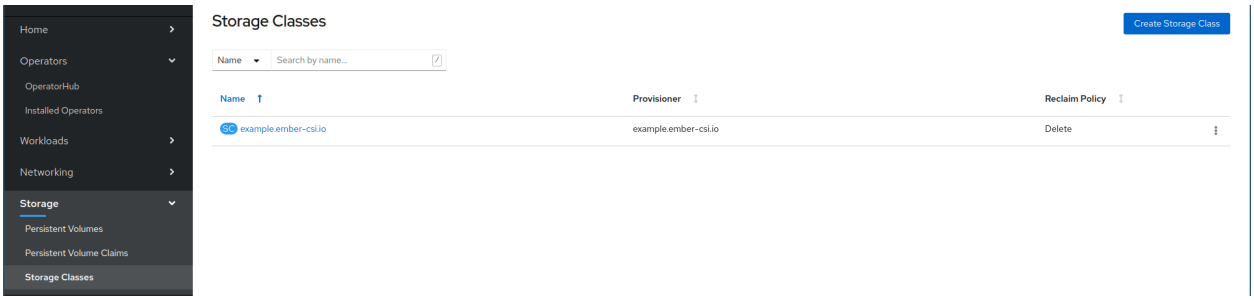
And a new *EmberStorageBackend* entity will be created. Don't wait for the *Status* to change, since it won't:



We can see that the deployment is complete going to *Stateful Sets*, *Daemon Sets*, and *Replica Sets* pages in the *Workloads* section to see that the deployed pods are running:

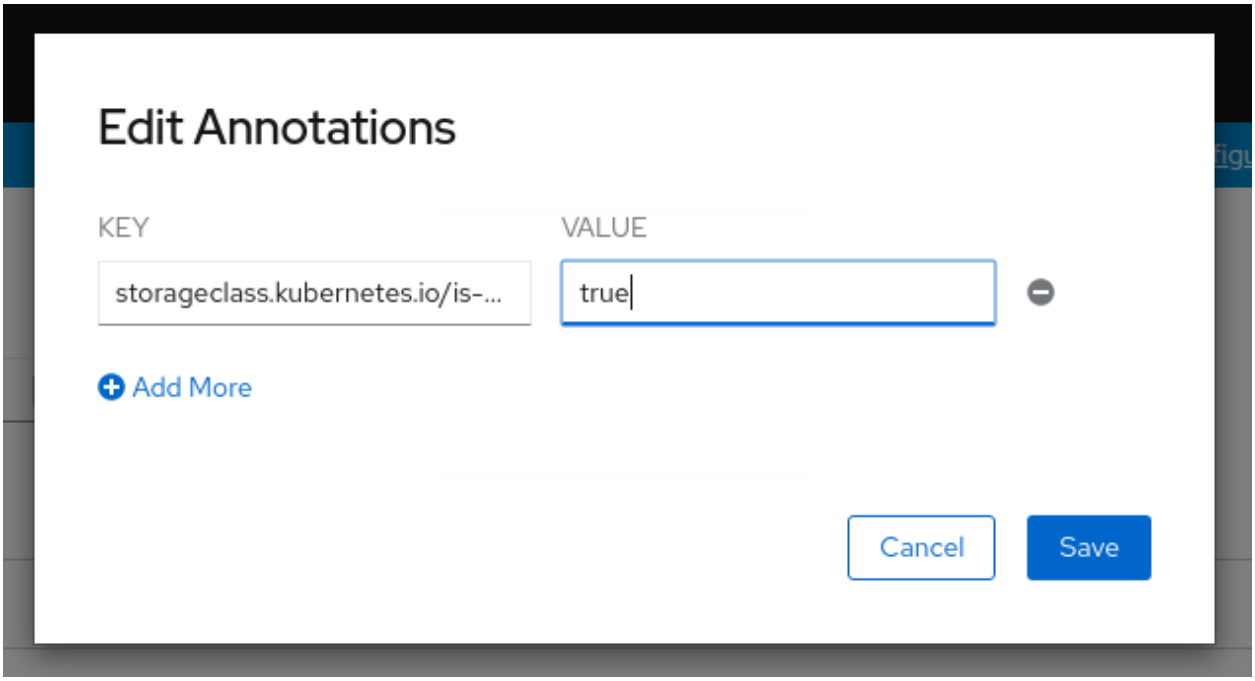


We can also check that a new *Storage Class* has been created in *Storage* > *Storage Classes*. The name of the new class will be *example.ember-csi.io* where *example* will be the name we gave to the *Storage Backend* in the form:



We can set this *Storage Class* as the default class by going to its actions and selecting *Edit Annotations*:

And then adding key *storageclass.kubernetes.io/is-default-class* with the value of *true*.



Warning: If we already have a default and we want to change it to this one, we'll need to modify the current default by removing the annotation or setting it to *false*.

If we have configured everything right we'll now be able to use our storage solution into OpenShift using the new `StorageClass` that was created by the operator. In the usage section there is information on how to use the new Storage Backend.

If you see problems in the new *Stateful*, *Daemon*, or *Replica Sets*, please refer to the troubleshooting guide for details on how to resolve installation issues.

CHAPTER 3

Usage

Now that we have completed the installation of Ember-CSI, we can manage our Storage Backend in our Container Orchestrator.

In this section examples will be provided both for the OpenShift Web Console and for the command line in the form of YAML manifests.

The same YAML manifests work on Kubernetes and OpenShift, the only difference is the command to invoke. For Kubernetes we'll use *kubectl'* and `oc` for OpenShift:

```
$ # On OpenShift
$ oc apply -f manifest.yaml

$ # On Kubernetes
$ kubectl apply -f manifest.yaml
```

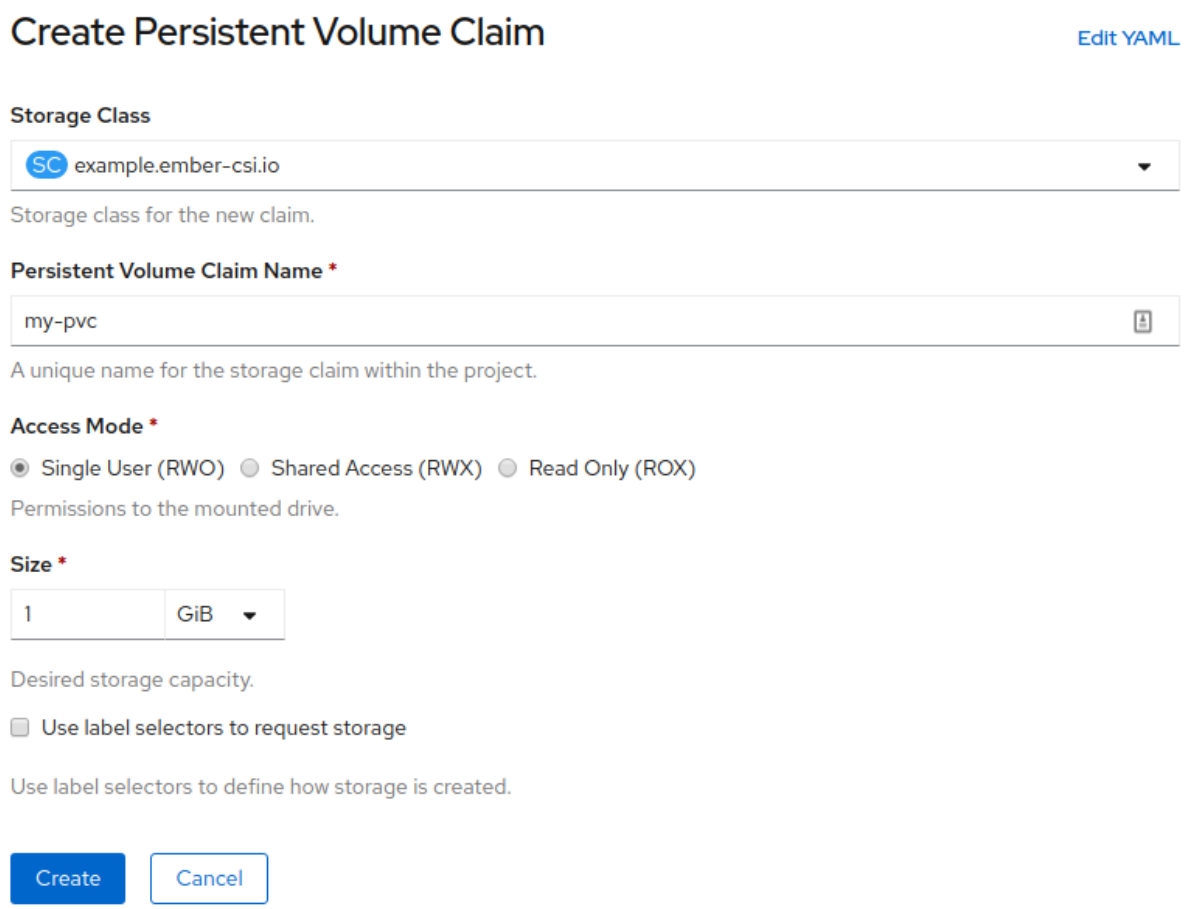
NOTE: In all the examples we'll assume we created the *Storage Backend* with the default name *example* using the Operator, and that the *Storage Class* automatically created is *example.ember-csi.io*

3.1 Volume creation

We can create 2 types of volumes, Block and File, and both are supported by Ember-CSI, but OpenShift forms don't allow specifying the type on creation, so they always default to File.

To create a volume we go to *Storage* > *Persistence VolumeClaims* and click on *Create Persistent Volume Clain*.

On the next page we must select the *Storage Class* created by the operator, give the *PVC* a unique name, select the *Access Mode* and the size.



Warning: Ember-CSI only supports the Shared Access (RWX) *Access Mode* for Block volumes.

Note: OpenShift doesn't support selecting the type of volume we want to create, so we'll have to use YAML if we want to create a Block volume.

To select the type of volume we want in our YAML we'll use the `volumeMode` parameter. Acceptable values are `Block` and `Filesystem`, the default being `Filesystem`.

Example of a *PVC* manifest using this default:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: example.ember-csi.io
```

Similar *PVC* example for a Block volume:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-block-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  resources:
    requests:
      storage: 3Gi
  storageClassName: example.ember-csi.io
```

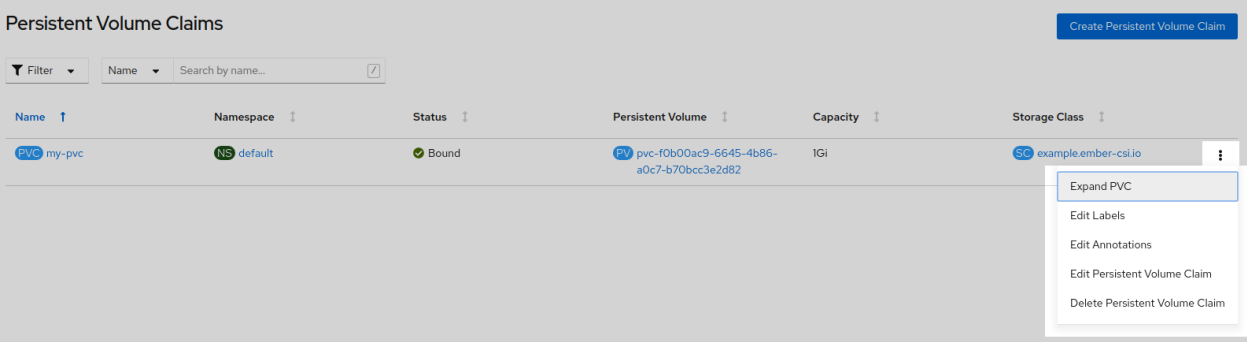
3.2 Using volumes

Using a dynamically created *PVC* is as easy as adding a `persistentVolumeClaim` parameter with the `claimName` in the `volumes` section of our manifest:

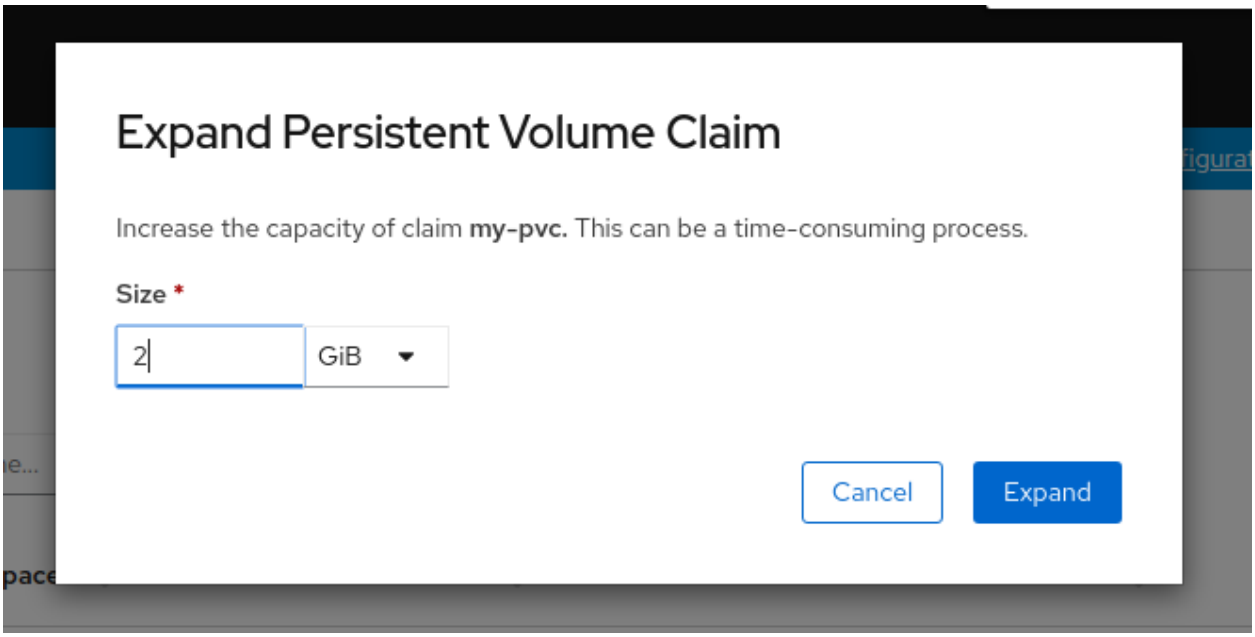
```
kind: Pod
apiVersion: v1
metadata:
  name: my-app
spec:
  containers:
    - name: my-frontend
      image: busybox
      volumeMounts:
      - mountPath: "/data"
        name: my-csi-volume
      command: [ "sleep", "1000000" ]

  volumes:
    - name: my-csi-volume
      persistentVolumeClaim:
        claimName: my-pvc
```

3.3 Expanding Volumes

We can expand already created volumes to have more space without losing existing data. The operation is called *expanding*, and it's very straightforward. In the web console we just go to the actions we can do in the *PVC* and select *Expand PVC*.



Then write the new size, that must be greater or equal than the existing size, and click on *Expand*.



When using the command line and a YAML manifest, we just need to modify the original contents with the new `storage` size, and it's important to use `apply` and not `create` on the command line:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: csi-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
  storageClassName: example.ember-csi.io
```

3.4 Volume cloning

Volume cloning is the process of creating a new volume with the same contents as the source volume.

The new volume must be greater or equal in size as the original one and the source volume must be specified in the

`dataSource` parameter, which is not available yet in the OpenShift Web Console, so we'll have to use YAML to do it:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: cloned-vol
spec:
  storageClassName: example.ember-csi.io
  volumeMode: Block
  dataSource:
    name: my-block-pvc
    kind: PersistentVolumeClaim
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

3.5 Snapshot creation

During the deployment phase the Operator also creates a *VolumeSnapshotClass* for our Storage Backend with the same name as the *StorageClass* so we can easily create snapshots.

Note: As of OpenShift 4.5 the Web Console doesn't have support for snapshots, but the necessary code is being merged in master, so it will most likely be available in OpenShift 4.6.

So we'll have to use a YAML manifest and use the `source` parameter to define the volume we want to snapshot.

```
apiVersion: snapshot.storage.k8s.io/v1beta1
kind: VolumeSnapshot
metadata:
  name: my-snapshot
spec:
  volumeSnapshotClassName: example.ember-csi.io
  source:
    persistentVolumeClaimName: my-block-pvc
```

3.6 Restoring a snapshot

To restore an already created snapshot we'll have to create a new volume and use our snapshot as its source.

The new volume must be of greater or equal size than the snapshot.

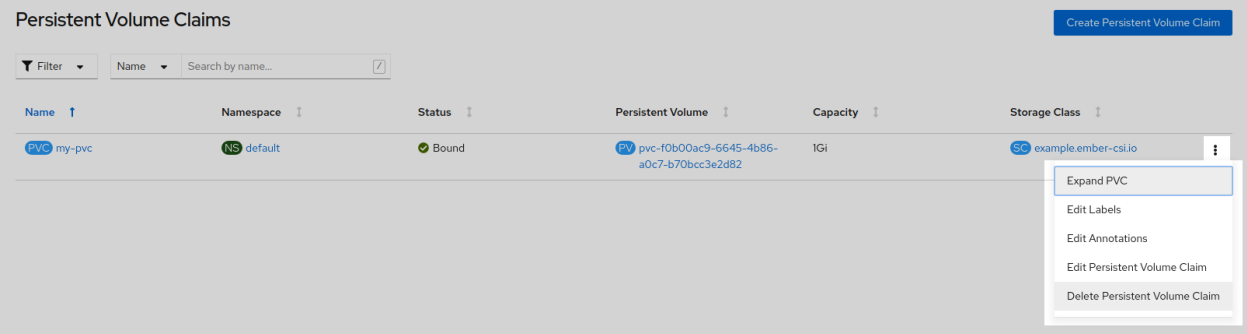
```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: restored-snapshot
spec:
  storageClassName: example.ember-csi.io
  dataSource:
    name: my-snapshot
```

```
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

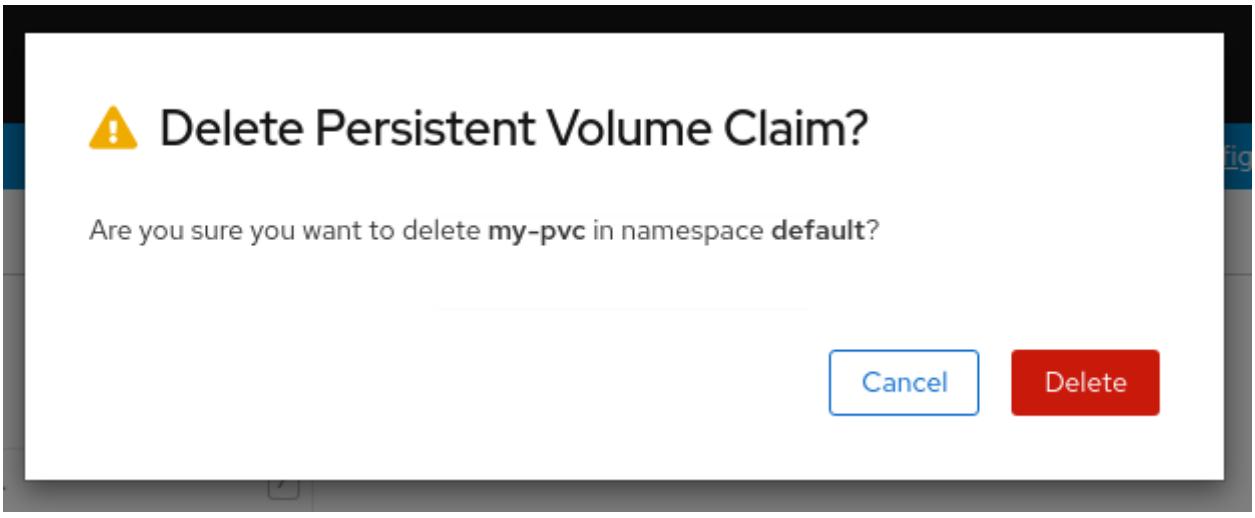
3.7 Volume deletion

The *Storage Class* created by the operator is defined with a `Delete` `ReclaimPolicy`, which means that once we delete the *PVC* the dynamically created *PV* will be deleted.

We can delete a volume in the OpenShift Web Console by going to *Storage* > *Persistence VolumeClaims* and look for the *PVC* we want to delete and in its actions we select *Delete Persistent Volume Claim*:



Deletion requires confirmation, so we'll have to click on the *Delete* volume:



Deleting the *PVC* from the command line can be done using the name:

```
$ oc delete pvc my-block-pvc
```

Or with the manifest we used to create it:

```
$ oc delete -f manifest.yaml
```

3.8 Snapshot deletion

Note: As of OpenShift 4.5 the Web Console doesn't have support for snapshots, but the necessary code is being merged in master, so it will most likely be available in OpenShift 4.6.

Deleting a *VolumeSnapshot* from the command line can be done using the name:

```
$ oc delete pvc my-snapshot
```

Or with the manifest we used to create it:

```
$ oc delete -f manifest.yaml
```

CHAPTER 4

Troubleshooting

The main tool used to investigate issues between Ember-CSI and the Orchestrator are OpenShift/Kubernetes status and logs.

Ember-CSI runs 2 types of services, one is the controller and the other is the node type. While the controller takes care of the management operations (create, delete, map/unmap, snapshots, etc.) the node mostly takes care of doing the local attach and detach on the hosts that are running the pods.

These services follow the CSI specification, exposing all their operations through a gRPC interface that needs to be translated into OpenShift/Kubernetes objects. The sidecars present in the Ember-CSI pods are responsible for the translation.

4.1 Status

The first thing we need to do when we encounter an issue is make sure that all the containers in the Ember-CSI pods, the driver container and the sidecars, are running and that their restart counts are not increasing.

Instead of looking at all the pod in our deployment we can use the fact that the operator adds the `embercsi_cr` label to filter for the pods of a specific backend:

```
$ # On OpenShift
$ oc get pod -n <cluster-namespace> -l embercsi_cr=<backend_name> -o wide

$ # On Kubernetes
$ kubectl get pod -n <cluster-namespace> -l embercsi_cr=<backend_name> -o wide
```

Or the pods for all the Ember-CSI backends:

```
$ oc get pod -n <cluster-namespace> -l embercsi_cr -o wide
```

When using an iSCSI or FC backend we need to make sure that the system daemons required for the connections are running and they are not reporting errors if we encounter issues on the following operations:

- Creating a volume from a source (volume or snapshot): On some drivers this is not a backend assisted operation, so the resources in the backend need to be accessed in the controller node.

- Creating or destroying a pod that uses an Ember-CSI volume:

If we are running the daemons as systemd services on baremetal, we can check them running:

```
$ systemctl status iscsid multipathd
$ sudo journalctl -u iscsid -u multipathd
```

On the other hand, if we are running the daemons in the foreground inside containers, we'll have to check the containers status and logs themselves.

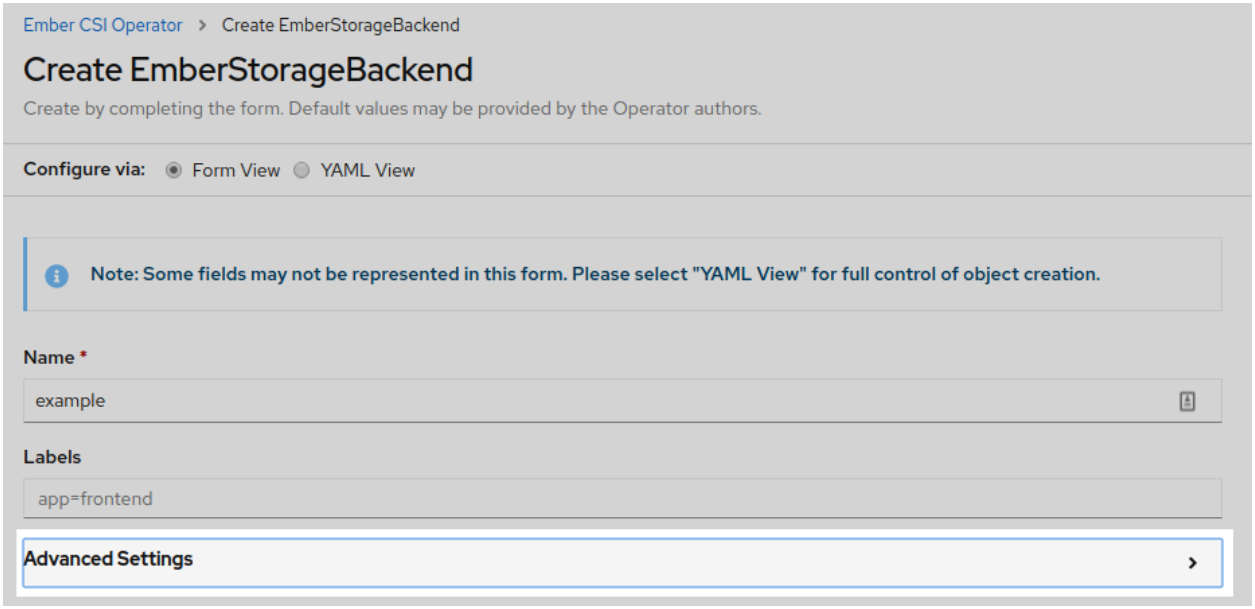
4.2 Logs

One of the most versatile tools to debug issues in general are the logs, and Ember-CSI is no different.

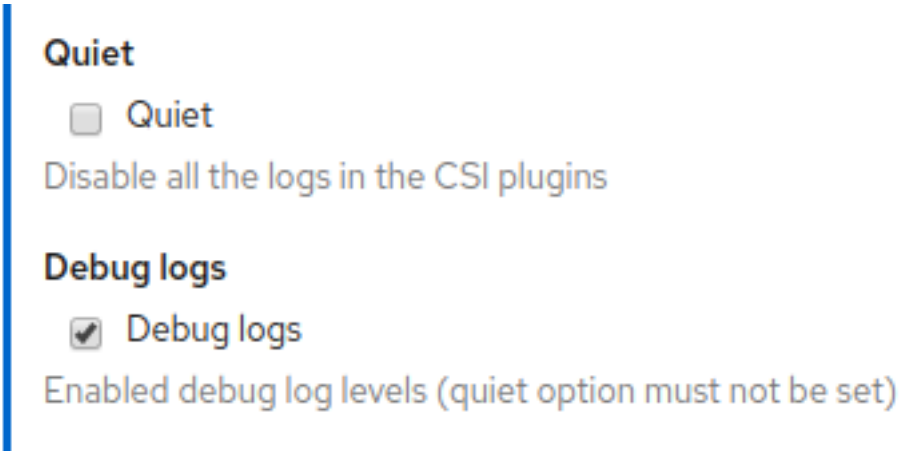
The logs we'll have to check will depend on the operations that are failing:

- If it's creating/deleting a volume or creating/deleting a snapshot, we should look into the Ember-CSI controller pod, primarily the driver container.
- Creating/destroying a pod that uses a volume is one of the most complex operations, and it requires the controller pod, the node pod, and the kubelet, so we'll have to look into all their logs.

By default Ember-CSI logs will be on *INFO* level and they can only be changed to *DEBUG* when creating the Storage Backend in the *Advanced Settings* section:



By setting the *Debug logs* checkbox:



4.3 CSC

When debugging issues on complex flows, it's very convenient to be able to test the individual tasks that form the flows. For that purpose the Ember-CSI has created containers with the `csc` tool for each of the CSI specs.

The `csc` tool allows us to execute specific CSI operations directly against an Ember-CSI service.

For example, we could run a create volume operation completely bypassing the Orchestrator. This way we could focus on the Ember-CSI code itself and the interactions with the storage solutions, removing the interactions with other elements such as OpenShift/Kubernetes scheduler and the sidecars.

Neither Kubernetes nor OpenShift allows adding containers to a running Pod, but there is an Alpha feature called Ephemeral Containers designed for debugging purposes that can do it.

We need to have the feature gate `EphemeralContainers` enabled in our Orchestrator. Specifically on the API, Scheduler, and Kubelet: `--feature-gates=EphemeralContainers=true`.

If it's enabled we can add an Ephemeral container with the `csc` command to our running pod.

For the following steps we'll assume we have used the name `example` as our Backend name.

First we check the CSI version that is using Ember-CSI:

```
$ oc describe pod example-controller-0|grep X_CSI_SPEC_VERSION
X_CSI_SPEC_VERSION:        1.0
```

Now that we know we are running CSI v1.0 we know the `csc` container we want to use: *embercsi/csc:v1.0.0*

With that we can write the `csc.json` file to add the Ephemeral Container:

```
{
  "apiVersion": "v1",
  "kind": "EphemeralContainers",
  "metadata": {
    "name": "example-controller-0"
  },
  "ephemeralContainers": [
    {
      "command": ["tail"],
      "args": ["-f", "/dev/null"],
      "image": "embercsi/csc:v1.0.0",
      "imagePullPolicy": "IfNotPresent",
      "name": "csc",
```

```
      "stdin": true,
      "tty": true,
      "terminationMessagePolicy": "File",
      "env": [ {"name": "CSI_ENDPOINT",
                "value": "unix:///csi-data/csi.sock"} ],
      "volumeMounts": [
          {
              "mountPath": "/csi-data",
              "mountPropagation": "HostToContainer",
              "name": "socket-dir"
          }
      ]
    }
  ]
}
```

And, assuming we don't have any other Ephemeral Containers, we add it by replace the current value:

```
$ oc replace --raw /api/v1/namespaces/default/pods/example-controller-0/
↪ephemeralcontainers -f csc.json
```

If we don't want to create a file we can do a one-liner by using `echo` a piping it to the `oc replace` command and setting the file contents to *stdin* with `-f -`.

Now that we have added the Ephemeral Container we can confirm it is running looking at the description of the controller pod and going to the `Ephemeral Containers` section and checking the `State`:

```
$ oc describe pod example-controller-0

...

Ephemeral Containers:
  csc:
    Container ID:  docker://
↪e52d25a53af77a6f660d171504aa9dc6c2c3d405a9af20451054fadba969c84a
    Image:         embercsi/csc:v1.0.0
    Image ID:      docker-pullable://embercsi/
↪csc@sha256:5433e0042725398b9398be1b73d43cc96c77893cf4b77cafca77001fa533cd29
    Port:          <none>
    Host Port:     <none>
    Command:
      sh
    State:          Running
      Started:      Thu, 13 Aug 2020 14:18:23 +0000
    Ready:          False
    Restart Count:  0
    Environment:
      CSI_ENDPOINT:  unix:///csi-data/csi.sock
    Mounts:
      /csi-data from socket-dir (rw)
```

When we have the shell container running we can run `csc` commands by attaching to the shell. For example to see the help:

```
$ oc attach -it example-controller-0 -c csc
If you don't see a command prompt, try pressing enter.
/ # csc
NAME
```

```
    csc -- a command line container storage interface (CSI) client

SYNOPSIS
    csc [flags] CMD

AVAILABLE COMMANDS
    controller
    identity
    node

Use "csc -h,--help" for more information
```

Warning: Just like with normal containers, once you add an Ephemeral Container to a pod you cannot remove it, so be sure to detach from the container and not `exit` the shell, or the container will no longer be running and you won't be able to use it (you cannot run `exec` on an Ephemeral Container).

Note: To detach from the `csc` container shell you must type the escape sequence *Ctrl+P* followed by *Ctrl+Q*.

4.4 CRDs

Ember-CSI uses OpenShift/Kubernets etcd service to store metadata of its resources in the form of CRDs. Existing CRDs are:

- Volume: Stores each volume's status as well as the information necessary to locate them in the storage solution.
- Snapshot: Stores the information necessary to locate each snapshot in the storage solution.
- Connection: Stores the connection information needed for a node to connect to a volume.
- KeyValue: Stores the connector information needed to map the volumes to the nodes on the storage solution.

These CRDs are just JSON dictionaries with all the information Ember-CSI needs to operate, and in some cases it can be useful to examine them to see internal information.