NVMesh CSI Driver Guide

1.4.1 — Last update: 11 March 2024

NVIDIA - Mellanox

Table of Contents

1. Copyright and Trademark Information

2. Preface

Excelero™ creates innovative, high performance storage solutions that accelerate business applications and deliver outstanding return on investment with the lowest cost of ownership. The NVMesh® software defined block storage product offers the performance of local server flash with the convenience, efficiency and redundancy of an all-flash-array. For details, go to: www.excelero.com.

This document describes the NVMesh CSI Driver for integration with Container Orchestration System (CO) e.g. Kubernetes. For more information on NVMesh refer to NVMesh User Guide.

AUDIENCE

The primary audience for this document is intended to be storage and/or application administration personnel responsible for installing and deploying the Excelero NVMesh product in a Container Orchestration environment.

NON-DISCLOSURE REQUIREMENTS

FEEDBACK

We continually try to improve the quality and usefulness of Excelero documentation. If you have any corrections, feedback, or requests for additional documentation, send an e-mail message to support@excelero.com

INFORMATION ABOUT THIS DOCUMENT

All information about this document including typographical conventions, references, and a glossary of terms can be found in the Document Reference Section.

3. Introduction

NVMesh CSI Driver is a Container Storage Interface (CSI) driver that allows Container Orchestration systems (COs) to use the NVMesh storage backend.
The driver allows COs to allocate, manipulate and remove NVMesh Persistent Volumes using the COs interface.

This document describes the Installation, Configuration and Usage of the NVMesh CSI Driver in all supported COs.

4. Kubernetes

NVMesh CSI driver is compatible with any Container Orchestration (CO) system that support the CSI spec. It may also be compatible with other provisioning systems that support CSI such as Openshift.

This section covers the usage with Kubernetes

4.1. Installation

Installation

```
kubectl create namespace nvmesh-csi

# k8s version 1.17 - 1.21
kubectl apply -f https://raw.githubusercontent.com/Excelero/nvmesh-csi-driver/v
1.4.1/deploy/kubernetes/deployment.yaml

# k8s versions  1.22 - 1.24
kubectl apply -f https://raw.githubusercontent.com/Excelero/nvmesh-csi-driver/v
1.4.1/deploy/kubernetes/deployment_k8s_1.22.yaml

# k8s version 1.25+
kubectl apply -f https://raw.githubusercontent.com/Excelero/nvmesh-csi-driver/v
1.4.1/deploy/kubernetes/deployment_k8s_1.25.yaml
```

Set Management Server Address

To let the CSI Driver know where your nvmesh-management server is, run the following on a machine with access to the cluster using kubectl and follow the instructions:
`bash <(curl -s https://raw.githubusercontent.com/Excelero/nvmesh-csi-driver/v1.4.1/deploy/kubernetes/scripts/set_mgmt_address.sh)`

This will update the nvmesh-csi-config ConfigMap and restart the nvmesh-csi PODs.

You can now skip to Quick Start example – Create Volume and POD

4.2. Configuration

Edit CSI Driver Config Map

To edit the config map use the command:

```
kubectl edit configmap -n nvmesh-csi nvmesh-csi-config
```

For any change to take effect you will need to restart the driver pods. To restart use the following command:
`kubectl rollout restart daemonset/nvmesh-csi-node-driver statefulset/nvmesh-csi-controller`

field name	type	description
`management.servers`	string	set to your `MANAGEMENT_SERVERS` configuration: `server-1.domain.com:4000` or `s-1.domain.com:4000,s-2.domain.com:4000`
`management.protocol`	`"http"` or `"https"`	The protocol used by the management server
`attachIOEnabledTimeout`	quoted int e.g `"30"`	The timeout in seconds for an attach to finish
`usePreempt`	`"true"` or `"false"`	if `"true"` the driver will always use the `preempt` flag for attach
`detachTimeout`	quoted int e.g `"90"`	The timeout in seconds for a detach to finish
`forceDetach`	`"true"` or `"false"`	if `"true"` the driver will always use the `force` flag for detach
`logLevel`	string	The log level of the CSI Driver. options: `"DEBUG"`, `"INFO"`, `"WARNING"`, `"ERROR"`
`sdkLogLevel`	string	The log level of the NVMesh SDK. options: `"DEBUG"`, `"INFO"`, `"WARNING"`, `"ERROR"`
`kubeClientLogLevel`	string	The log level of the k8s client. options: `"DEBUG"`, `"INFO"`, `"WARNING"`, `"ERROR"`
`printStackTraces`	`"true"` or `"fa`	if `"true"` will print stack traces on errors

	`lse"`	
`csiConfigMapName`	string	The name of this `configmap`
`topology`	string	The topology configuration. see more details in Multiple NVMesh Clusters
`topologyConfigMapName`	string	The name of the topology `configmap` see more details in Multiple NVMesh Clusters

Edit Management Server Username and Password

```
kubectl edit secret -n nvmesh-csi nvmesh-credentials
```

Edit `username` and `password` to your management server credentials configuration.

> Secrets in Kubernetes must be in base64 format

For example, use:

```
echo -n 'admin@excelero.com' | base64
```

and

```
echo -n 'admin' | base64
```

to get the username and password in base64.

For more info visit: Kubernetes Docs – Convert your secret data to a base-64 representation.

4.3. Uninstall

Uninstall the nvmesh-csi-driver by simple running the kubectl with the `delete` command:

```
# k8s version 1.17 - 1.21
kubectl delete -f https://raw.githubusercontent.com/Excelero/nvmesh-csi-driver/v
1.4.1/deploy/kubernetes/deployment.yaml

# k8s versions  1.22 - 1.24
kubectl delete -f https://raw.githubusercontent.com/Excelero/nvmesh-csi-driver/v
1.4.1/deploy/kubernetes/deployment_k8s_1.22.yaml

# k8s version 1.25+
kubectl delete -f https://raw.githubusercontent.com/Excelero/nvmesh-csi-driver/v
1.4.1/deploy/kubernetes/deployment_k8s_1.25.yaml

kubectl delete namespace nvmesh-csi
```

4.4. Usage

This topic describes how to use the NVMesh CSI Driver in Kubernetes.

4.4.1. Quick start example – PVC and POD

This quick start guide walks you through creating a BlockVolume using the NVMesh CSI Driver and using this volume from a POD.

Prerequisite

Before you continue, please make sure you have already installed the NVMesh CSI Driver on your cluster.

Create a PVC

Create a volume using the following PVC yaml:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteMany
  volumeMode: Block
  resources:
    requests:
      storage: 5Gi
  storageClassName: nvmesh-concatenated
```

Run the following command and check the output to make sure your volume was created successfully:

```
$kubectl get pvc
NAMESPACE   NAME        STATUS   VOLUME                                     CAPAC
ITY   ACCESS MODES   STORAGECLASS          AGE
default     block-pvc   Bound    pvc-2ec86fdd-f656-4810-9a03-54fcd668a705   5G
i        RWX            nvmesh-concatenated   2s
```

Go to your NVMesh-Managment GUI. You should be able to see that a new volume was created.

Create a POD

Create a POD using the following PVC yaml:

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: block-volume-consumer-pod
  labels:
    app: block-volume-consumer-test
spec:
  containers:
    - name: block-volume-consumer
      image: excelero/qguide_block_volume_consumer
      args: ["/dev/my_block_dev"]
      volumeDevices:
        - name: block-volume
          devicePath: /dev/my_block_dev
  volumes:
    - name: block-volume
      persistentVolumeClaim:
        claimName: block-pvc
```

Run the following command. Check the output to make sure your pod was created successfully:

```
$ kubectl get pod block-volume-consumer-pod
NAME                        READY   STATUS    RESTARTS   AGE
block-volume-consumer-pod   1/1     Running   0          20s
```

Check the logs:

```
$ kubectl logs block-volume-consumer-pod
Writing to file /dev/my_block_dev
Read 15 bytes: "Excelero NVMesh"
- Sleeping
- Sleeping
```

The following indicates that the Container in the pod had successfully written and read from the block device `/dev/my_block_dev`

4.4.2. Creating a PersistentVolumeClaim

In Kubernetes, a `PersistentVolumeClaim` (PVC) is a request for storage by a user.

Let's look at an example of a PVC yaml, and then describe the fields relevant for NVMesh and their options. For more information on PersistentVolumeClaims, see K8s Docs – PersistentVolumesClaims.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteMany
  volumeMode: Block
  resources:
    requests:
      storage: 10Gi
  storageClassName: nvmesh-concatenated
```

Access Modes (accessModes)

The Kubernetes `accessMode` defines a **per Node semantics** of how the user wants to access the volume. The following values are accepted:

- `ReadWriteOnce` – the volume can be mounted as read-write by a single node.
- `ReadOnlyMany` – the volume can be mounted read-only by many nodes.
- `ReadWriteMany` – the volume can be mounted as read-write by many nodes.

Volume Mode (volumeMode)

The `volumeMode` field controls which type of volume will be created. accepted values are:

- `Block` – Will create a raw block NVMesh volume.
- `FileSystem` – Will create a block NVMesh volume and upon first attach the volume will be formatted into a FileSystem according to the FileSystem defined in the StorageClass. See Important Nodes and Known Issues for more info on FileSystem volumes limitations.

For more information on `volumeMode` please refer to K8s Docs – PersistentVolumesClaims

Request Storage (resources.requests.storage)

This enables entering the amount of Storage to be provisioned for the requested volume.

A value of `100Gi` will create a 100GiB Volume in NVMesh.

Storage Class Name (storageClassName)

This is the name of the StorageClass object in Kubernetes.
It will tell Kubernetes that NVMesh is the storage backend as well as declare the volume type and its parameters.
After installing the nvmesh-csi-driver, default StorageClass objects for each of the default NVMesh Volume Provisioning Groups (VPGs) will be created.
Following is the list of default StorageClass names and their corresponding VPG in NVMesh:

StorageClass name	NVMesh VPG
`nvmesh-concatenated`	DEFAULT_CONCATENATED_VPG
`nvmesh-raid0`	DEFAULT_RAID_0_VPG
`nvmesh-raid1`	DEFAULT_RAID_1_VPG
`nvmesh-raid10`	DEFAULT_RAID_10_VPG
`nvmesh-ec-dual-target-redundancy`	DEFAULT_EC_DUAL_TARGET_REDUNDANCY_VPG
`nvmesh-ec-single-target-redundancy`	DEFAULT_EC_SINGLE_TARGET_REDUNDANCY_VPG

See more about StorageClasses here.

4.4.3. Multiple NVMesh Clusters & Topology

Introduction

The NVMesh CSI Driver Topology feature allows a single CSI driver to manage multiple clusters of NVMesh within a single Kubernetes environment.
The driver topology feature ensures that each pod using a NVMesh-based PVC will only be scheduled on nodes where the volume is accessible from the NVMesh client.

When the topology feature is configured, each NVMesh cluster will be represented as an NVMesh CSI zone. The driver automatically adds a label on each node in the format `nvmesh-csi.excelero.com/zone=<zone name>` to have Kubernetes associate each node with a cluster or zone.

The configuration of zones is configured by the administrator in the `nvmesh-csi-driver-config` ConfigMap. The driver will discover all nodes for z given zone by querying the NVMesh management servers configured for that zone and will save this topology in a new ConfigMap named `nvmesh-csi-topology`, This ConfigMap should not be modified by the user. When a volume is created, the driver will add `nodeAffinity` to the `PersistentVolume` with the zone label to let the Kubernetes scheduler know that all future pods using this PVC should be scheduled only on nodes in the same zone as the NVMesh cluster where the volume was provisioned.

Configuration

To inform the CSI driver of the available zones add the `topology` field to the `nvmesh-csi-driver-config` ConfigMap.
Following is an example with a list of all available options.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: nvmesh-csi-driver-config
data:
  management.protocol: https
  management.servers: 10.0.1.117:4000
  attachIOEnabledTimeout: "30"
  topology: |-
    {
       "zones": {
          "zone_A": {
             "management": {
                "servers": "worker1.domain.com:4000"
             }
```

```
        },
        "zone_B": {
           "management": {
              "servers": "worker4.domain.com:4000"
           }
        }
     }
  }
```

The topology field is a JSON with a single `zones` key, which contains the configuration for each zone. Each key in the `zones` object is a name of a zone and the value provides the zone configuration parameters.

For each zone configuration, the following fields are available:

Field	Description
`management`	Configuration for the management server in this specific zone
`management.servers`	A comma-separated list of management servers addresses in the format `address:port`, for instance `management-1:4000,management-2:4000`
`management.protocol`	The management server protocol, i.e. “http” or “https”
`management.user`	The management user to login with, for instance “admin@excelero.com”
`management.password`	The management password, for instance “admin”

Creating Volumes and Pods

Create a PVC and a Pod

Create a `StorageClass` with `volumeBindingMode: WaitForFirstConsumer`.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: nvmesh-with-topology
provisioner: nvmesh-csi.excelero.com
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
```

```
parameters:
  vpg: DEFAULT_CONCATENATED_VPG
```

Create a PVC using this `StorageClass`

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: topology-volume0
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 1Gi
  storageClassName: nvmesh-wait-for-consumer
```

Create a Pod that uses the PVC

```
apiVersion: v1
kind: Pod
metadata:
  name: topology-pod0
spec:
  serviceAccountName: topology-aware
  containers:
    - name: nginx
      image: gcr.io/google_containers/nginx-slim:0.8
      ports:
      - containerPort: 80
        name: web
      volumeMounts:
      - name: www
        mountPath: /usr/share/nginx/html
  volumes:
    - name: www
      persistentVolumeClaim:
        claimName: topology-volume0
```

Assign the PVC / Pod to a zone using a StorageClass with the topology field

To create volumes on a specific NVMesh cluster, create a `StorageClass` with the `allowedTopologies` field.
When a PVC is created from a StorageClass with this field, the CSI driver will create the volume on the desired zone.

Multiple allowedTopologies

If multiple zones are allowed, as in the example below, the CSI driver will randomly pick one of the zones and create the volume on that zone.
The `PersistentVolume` will then be accessible only on the selected zone and every pod with the same PVC will only be scheduled to that selected zone.
Different PVCs created from the same storageClass may be in different zones.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: nvmesh-with-topology
provisioner: nvmesh-csi.excelero.com
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
parameters:
  vpg: DEFAULT_CONCATENATED_VPG
allowedTopologies:
- matchLabelExpressions:
  - key: nvmesh-csi.excelero.com/zone
    values:
    - zone_A
    - zone_B
```

Assign a PVC or Pod to a zone using the Pod's nodeAffinity

It is possible to set the nodeAffinity directly on the pod. The PVC and the pod will then be created in the desired zone. In this case, the PVC should use a StorageClass with `volumeBindingMode: WaitForFirstConsumer`.

```
apiVersion: v1
kind: Pod
metadata:
  name: topology-pod0
spec:
  serviceAccountName: topology-aware
```

```
spec:
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: nvmesh-csi.excelero.com/zone
          operator: In
          values:
          - zone_A
          - zone_B
containers:
  - name: nginx
    image: gcr.io/google_containers/nginx-slim:0.8
    ports:
    - containerPort: 80
      name: web
    volumeMounts:
    - name: www
      mountPath: /usr/share/nginx/html
volumes:
  - name: www
    persistentVolumeClaim:
      claimName: topology-volume0
```

For a more complex example with StatefulSet, Multiple Zone and antiAffinity on zones, see Topology-Aware Volume Provisioning in Kubernetes

PVC with volumeBindingMode: Immediate

When a PVC with `volumeBindingMode: Immediate` is created, the NVMesh CSI Driver will randomly pick a zone and provision the volume on that zone.
All subsequent pods using this PVC will be scheduled to this zone.

> `volumeBindingMode: WaitForFirstConsumer` should be preferred as this will allow the Kubernetes scheduler to schedule the pod to the most fitting node taking into account the load on nodes and their capabilities, such as network, CPU, memory etc. The PVC will then be provisioned on the zone where the first pod was scheduled.

References

For additional details on VolumeBindingMode, see k8s Documentation – VolumeBindingMode

For additional details on AllowedTopologies, see k8s Documentation – AllowedTopologies

4.4.4. Important Notes and Known Issues

AccessMode

The Kubernetes `AccessMode` field in a PVC can receive the following values:
(reference K8s Docs – Volume AccessMode)

- `ReadWriteOnce` – the volume can be mounted as read-write by a single node
- `ReadOnlyMany` – the volume can be mounted read-only by many nodes
- `ReadWriteMany` – the volume can be mounted as read-write by many nodes

> Kubernetes `AccessModes` as defined today, only describe node attach (not pod mount) semantics. For example when using `AccessMode: ReadWriteOnce` The NVMesh CSI Driver will allow the attach to happen only on one node BUT **does not guarantee** that 2 pods running on the same node will not access the volume at the same time.

FileSystem Volumes

When creating a `FileSystem` Volume the CSI Driver currently supports only non-shared File Systems (ext4 and xfs)
This means that the user should make sure that no more than one POD is writing to the Volume at the same time, and multiple readers might not have the most updated data.
About using the PVC `AccessMode` field please see below.

To deploy any other file system, please create a BlockVolume and deploy the file system after the volume was created.

> When `FileSystem` Volume is used Make sure you have configured the consuming PODS to have only one writer at a time. having multiple writers might cause the attach process to hang making the volume unusable.

4.4.5. Examples

This section covers examples of creating Kubernetes objects that use NVMesh Storage backend.

4.4.5.1. File System Volume

The driver deployment creates storage-classes that correspond to each of the NVMesh default VPGs.

The following storage classes will appear under namespace “nvmesh-csi”:

- `nvmesh-concatenated`
- `nvmesh-raid0`
- `nvmesh-raid1`
- `nvmesh-raid10`
- `nvmesh-ec`

Before creating a FileSystem volume, see Important Notes and Known Issues.

Create a PersistentVolumeClaim of type RAID1

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nvmesh-raid1
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 3Gi
  storageClassName: nvmesh-raid1
```

- This will default to a FileSystem Volume with ext4.

Create a Storage-Class for volumes with the XFS FileSystem

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: nvmesh-xfs-class
provisioner: nvmesh-csi.excelero.com
allowVolumeExpansion: true
volumeBindingMode: Immediate
parameters:
```

```
  vpg: DEFAULT_CONCATENATED_VPG
  fsType: xfs
```

Create a volume from the XFS Storage-Class

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nvmesh-xfs-volume
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 3Gi
  storageClassName: nvmesh-xfs-class
```

4.4.5.2. Block Volume

Create a Raw Block Volume (Kubernetes 1.14 or higher)

See section Important Notes and Known Issues.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteMany
  volumeMode: Block
  volumeBindingMode: Immediate
  resources:
    requests:
      storage: 3Gi
  storageClassName: nvmesh-concatenated
```

4.4.5.3. Storage Class

In Kubernetes, a `StorageClass` provides a way for administrators to describe the “classes” of storage they offer.
For NVMesh, different StorageClasses could describe different type of volumes that will be created by the NVMesh backend. (e.g different RAID Levels).

Let’s look at an example of a StorageClass yaml and then describe the fields relevant to NVMesh and their options.
For more information on StorageClass, see K8s Docs – StorageClass.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: nvmesh-raid10
provisioner: nvmesh-csi.excelero.com
allowVolumeExpansion: true
volumeBindingMode: Immediate
mountOptions:
  - debug
parameters:
  vpg: DEFAULT_RAID_10_VPG
```

Provisioner

The `provisioner` field determines which storage backend driver / plugin will be used for provisioning the volume.
To create an NVMesh volume, this field must be set to: `nvmesh-csi.excelero.com`

Allow Volume Expansion (allowVolumeExpansion)

The field `allowVolumeExpansion` controls whether the volume should be expandable or not.
For more info, see K8s Docs – Allow Volume Expansion.

Volume Binding Mode (volumeBindingMode)

The `volumeBindingMode` field controls volume provisioning timing. The following values are accepted:

- `Immediate`: This is the default if omitted. If this value is set, dynamic provisioning occurs once the PersistentVolumeClaim is created.
- `WaitForFirstConsumer`: If this value is set, volume provisioning will be delayed until a Pod using the PersistentVolumeClaim is created.

For more info on volumeBindingMode, see K8s Docs – VolumeBindingMode

Mount Options (mountOptions)

The `mountOptions` field enables setting the options passed to the mount command (`mount -o <options>`) and a special key allows to set the mount permissions.

To set the mount permissions, for example to 777, use:

```
mountOptions:
  - nvmesh:permissions=777
```

Mount options are not validated on either the class or PV. If a mount option is invalid, the PV mount fails.

Parameters

The `parameters` field is a structure used to define NVMesh specific parameters. The following parameters are accepted:

field name	type	description
`fsType` Deprecated	choice ext4,xfs	Deprecated, see `csi.storage.k8s.io/fstype` below
`csi.storage.k8s.io/fstype`	choice ext4,xfs	When a PVC has `volumeMode: FileSystem`, the `fsType` field will determine which FileSystem type will be deployed on the volume. Accepted values are: `ext4`, `xfs`
`mkfsOptions`	string, optional	Flags and extended options to pass to mkfs command when creating a Filesystem. For available options, see the documentation of `mkfs.ext4` or `mkfs.xfs` e.g: `mkfsOption: -b 4096`
`vpg`	string, optional	The name of the NVMesh Volume Provisioning Group (VPG) as defined in the NVMesh Management.
`raidLevel`	string, optional	The volume type, allowed values are: `concatenated`, `raid0`, `raid1`, `raid10` and `ec`.
`diskClasses`	list, optional	Limit volume allocation to specific diskClasses, defaults to None.
`serverClasses`	list, optional	Limit volume allocation to specific serverClasses, defaults to None.

`limitByDisks`	list, optional	Limit volume allocation to specific disks, defaults to None.
`limitByNodes`	list, optional	Limit volume allocation to specific nodes, defaults to None.
`encryption: dmcrypt`	string, optional	Provision encrypted volumes – must equal `dmcrypt`
`csi.storage.k8s.io/node-stage-secret-name`	string, optional	Encryption key Secret object name
`csi.storage.k8s.io/node-stage-secret-namespace`	string, optional	Encryption key Secret object namespace
`dmcrypt/type`	string, optional	select LUKS header type.
`dmcrypt/cipher`	string, optional	select encryption cipher.

If `raidLevel` is defined, the following parameters are accepted according to the selected `raidLevel`:

- If raidLevel is `raid0` or `raid10`
  - `stripeSize` (integer, optional) – number in blocks of 4k, i.e. stripeSize:32 = 128k, optional, defaults to 32.
  - `stripeWidth` (integer, optional) – number of disks to use, defaults to 2.

- If raidLevel is `ec`
  - `dataBlocks` (integer, optional) – number of disks to use, defaults to 8.
  - `parityBlocks` (integer, optional) – number of disks to use, defaults to 2.
  - `protectionLevel` (string, optional) – protection level to use, allowed values are `Full Separation`, `Minimal Separation`, `Ignore Separation` defaults to `Full Separation`.

All `integer` values must be wrapped with quotes for Kubernetes to accept the yaml. i.e `stripeWidth: "2"`.

For more info on the specific fields, their purpose and allowed value range, see the NVMeshSDK Documentation. To open NVMeshSDK documentation, go to your NVMesh Management Server and on the top right corner click Docs > SDK.

4.4.5.4. Using a Custom VPG

Create a VPG in the NVMesh Management software named `your_custom_vpg`.
Create a Storage-Class that will refer to the VPG we just created.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: nvmesh-custom-vpg
provisioner: nvmesh-csi.excelero.com
allowVolumeExpansion: true
volumeBindingMode: Immediate
parameters:
  vpg: your_custom_vpg
```

Create a volume from the Storage-Class.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nvmesh-custom-vpg-volume
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 15Gi
  storageClassName: nvmesh-custom-vpg
```

4.4.5.5. Static Provisioning

For Static Provisioning, we have an existing NVMesh volume that was created outside the CSI scope and want to consume it in Kubernetes.

By default, when creating a PVC from an NVMesh StorageClass, the CSI Driver will create a new NVMesh Volume and a new PersistentVolume in Kubernetes will be created to describe the new volume. This is called **Dynamic Provisioning**.
However, If you have an existing NVMesh Volume, possibly already populated with data, to consume it in Kubernetes, you will need to use **Static Provisioning**.

Following is an example of Static Provisioning:

- This example is also available in the github repo under docs/examples/static-provisioning.yaml.

Create NVMesh Volume

Create a volume in the NVMesh Management software with the following attributes:

- Name: vol-1
- Capacity: 5Gi
- Raid Type: RAID10 (you could use the DEFAULT_RAID_10_VPG)

Create a PersistentVolume in Kubernetes

Create a PersistentVolume in Kubernetes to represent the volume already defined in the NVMesh:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: name-in-k8s
spec:
  accessModes:
  - ReadWriteMany
  - ReadWriteOnce
  - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  capacity:
    storage: 5Gi
  volumeMode: Block
  storageClassName: nvmesh-raid10
  csi:
    driver: nvmesh-csi.excelero.com
```

```
    volumeHandle: vol-1
```

Relevant fields info:

`metadata.name` is the name that this PV will have in Kubernetes.

`spec.csi.driver` must be set to `nvmesh-csi.excelero.com`.

`spec.csi.volumeHandle` is the name of the volume in NVMesh.

`persistentVolumeReclaimPolicy`, by setting this field to `Retain` we let Kubernetes know this PersistentVolume should not be deleted when the bounded PVC is deleted.

`accessModes`, note that in this example we allowed all Access Modes, but you can choose any sub-set of these 3 options.

Create a PersistentVolumeClaim

Create a PVC that will be bound to the PV just created.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-1
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  resources:
    requests:
      storage: 5Gi
  storageClassName: nvmesh-raid10
```

Create a Pod that uses the Volume

Run a pod that will mount this volume and use it:

- This pod specifically does nothing with the volume, but you could get a shell to the running container and explore or run IO on the volume.
- The volume is available inside the pod under /vol.

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: pod-1
spec:
      containers:
      - name: c-1
        image: alpine
        command: ["/bin/sh", "-c", "echo hello ; while true ; do wait 1; done"]
        volumeDevices:
            - name: vol
              devicePath: /vol
      restartPolicy: Never
      volumes:
        - name: vol
          persistentVolumeClaim:
            claimName: pvc-1
```

4.4.5.6. Read Only Volume

How to Create a Read-Only NVMesh Volume & Populate it with Data

This example describes how to create an NVMesh volume for use as a ReadOnlyMany Persistent Volume. We will go over creating a Volume, populating it with data and then turn it into a ReadOnlyMany Volume.

> The following example uses volumeMode: Filesystem but the same applies for volumeMode: Block.

Create a Storage Class with reclaimPolicy: Retain.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: nvmesh-concatenated-retained
provisioner: nvmesh-csi.excelero.com
parameters:
  # set here the desired VPG
  vpg: DEFAULT_CONCATENATED_VPG
# set reclaimPolicy to retain so that the PV will not be deleted when it's PVC i
s deleted
reclaimPolicy: Retain
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

Create a PVC for populating the volume with data.
This will create a new volume with `accessMode ReadWriteOnce` so we can write data into the volume.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: claim-populate-vol-with-data
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

```
  storageClassName: nvmesh-concatenated-retained
  volumeMode: Filesystem
```

Create a pod to write the data to the volume.

Example: This pod will create a file with data.

```
apiVersion: v1
kind: Pod
metadata:
  name: populate-vol-with-data
spec:
  restartPolicy: OnFailure
  containers:
    - name: write-to-volume
      image: centos:7
      command: ["/bin/bash", "-c", "echo some-data > /data/data.txt"]
      volumeMounts:
        - name: data-volume
          mountPath: /data/
  volumes:
    - name: data-volume
      persistentVolumeClaim:
        claimName: claim-populate-vol-with-data
```

When the Pod is finished, delete the pod and also delete any workload using the PVC.

```
kubectl delete pod populate-vol-with-data
```

Delete the PVC.
As we used the storage-class with **reclaimPolicy: retain**, the PV will not be deleted by this action.

```
kubectl delete pvc claim-populate-vol-with-data
```

Edit the PersistentVolume Object.

Run this to find the PV created by the Claim:

kubectl get pv -o=custom-columns=NAME:.metadata.name,PVC:.spec.claimRef.name | grep claim-populate-vol-with-data

Edit the PV object by running:

```
kubectl edit pv <pv name>
```

Perform the following changes:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pvc-f89b81c9-1c23-40c0-b3a7-eb70525c25ea
spec:
  capacity:
    storage: 1Gi
  csi:
    ...
  accessModes:
    # change ReadWriteOnce to ReadOnlyMany
    # - ReadWriteOnce
    - ReadOnlyMany
  # Remove claimRef so that the PV can be bounded again to a new PVC
  #claimRef:
  #  ...
  persistentVolumeReclaimPolicy: Retain
  storageClassName: nvmesh-concatenated-retained
  volumeMode: Filesystem
```

Create a PVC with ReadOnlyMany.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: claim-rom
spec:
  accessModes:
    - ReadOnlyMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: nvmesh-concatenated-retained
  volumeMode: Filesystem
```

Create a pod to read the data.

Example Pod:
This pod will read the `data.txt` file and then try to delete the file printing the exit code (reading should succeed and deletion should fail).

```
apiVersion: v1
kind: Pod
metadata:
  name: read-data
spec:
  restartPolicy: OnFailure
  containers:
    - name: read-data
      image: centos:7
      cmd: ["/bin/bash", "-c" ,"cat /data/data.txt ; rm /data/data.txt; echo exi
t_code=$?"]
      volumeMounts:
        - name: data-volume
          mountPath: /data/
  volumes:
    - name: data-volume
      persistentVolumeClaim:
        claimName: claim-rom
```

4.4.5.7. Encrypted Volumes

Creating Encrypted Volumes

NVMesh CSI Driver uses dmcrypt to create encrypted volumes.

The YAMLs for the following example are available at: dmcrypt examples on GitHub

Create a secret that will hold the key

> The password must be at least 8 characters, should be complex and not include systematic characters like “123456”, “abc” , “qaz”

The key field name must be `dmcryptKey`

```
apiVersion: v1
kind: Secret
metadata:
  name: dmcrypt-example-key
data:
  # echo "my-dm-crypt-key" | base64
  dmcryptKey: bXktZG0tY3J5cHQta2V5Cg==
```

Create a StorageClass

Parameters explanation:
`encryption: dmcrypt` – required, Use encryption
`csi.storage.k8s.io/node-stage-secret-name: dmcrypt-example-key` – required, The k8s Secret object name
`csi.storage.k8s.io/node-stage-secret-namespace: nvmesh-csi` – required, The k8s Secret object namespace

`dmcrypt/type: "luks2"` – optional, change the LUKS header type
`dmcrypt/cipher: "aes-xts-plain64"` – optional, change the cipher

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: encrypted-nvmesh-xfs
provisioner: nvmesh-csi.excelero.com
```

```
allowVolumeExpansion: true
volumeBindingMode: Immediate
parameters:
  vpg: DEFAULT_CONCATENATED_VPG
  csi.storage.k8s.io/fstype: xfs
  encryption: dmcrypt
  csi.storage.k8s.io/node-stage-secret-name: dmcrypt-example-key
  csi.storage.k8s.io/node-stage-secret-namespace: nvmesh-csi
# optional parameters:
  dmcrypt/type: "luks2"
  dmcrypt/cipher: "aes-xts-plain64"
```

Create a PVC

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-encrypted-xfs
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 10Gi
  storageClassName: encrypted-nvmesh-xfs
```

Create a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-encrypted-volume
spec:
  containers:
    - name: cento7
      image: centos:7
      command:
        - /bin/bash
        - '-c'
        - '--'
      args:
```

```
      - "while true; do sleep 1 & wait $!; done;"
    volumeMounts:
    - name: vol1
      mountPath: /mnt/vol1
volumes:
  - name: vol1
    persistentVolumeClaim:
      claimName: pvc-encrypted-xfs
```

5. Document Reference

Typographical Conventions

Throughout this document, the following typographical conventions are followed:

Style	Meaning
bold text	The name of an Excelero software component or technology
`text`	A file name, command or configuration text that can be utilized in a Linux terminal/shell, file or as a URL
term in italics	Generally, a term being used in specific relation to an element in the NVMesh

Definitions

Throughout this document, these terms have the following meanings:

Term	Definition
Management Server	The server(s), or OS image(s) running the **management module** software
Target Node/ Target	A physical server containing one or more NVMe SSDs running the **storage target module**
Client Node/ Client	An OS image instance running the **block storage client** software
Converged Node	A *target node* that is also running the **block storage client** software
Logical Volume/ Volume	A logical block device defined with the **NVMesh management module** that can be attached to *client nodes*
RDDA	Remote Direct Drive Access. Excelero's patented low-latency and CPU bypass transport technology.
TOMA	**To**pology **Ma**nager. The **storage target module** component that handles error detection and volume rebuild activities.

6. Versions

Version Compatibility

NVMesh CSI Driver	Kubernetes	NVMesh
1.4.1	1.17-1.25	2.6 – 2.7
1.4.0	1.17-1.25	2.6 – 2.7
1.3.0	1.17-1.25	2.6
1.2.6	1.17-1.25	2.2-2.6
1.2.5	1.17-1.25	2.2-2.6
1.2.4	1.17-1.25	2.2-2.6
1.2.3	1.17-1.25	2.2-2.6
1.2.2	1.17-1.25	2.2-2.6
1.2.1	1.17-1.25	2.2-2.6
1.2.0	1.17-1.25	2.2-2.6
1.1.7	1.17-1.25	2.2
1.1.6	1.17-1.21	2.2
1.1.5	1.17-1.21	2.2
1.1.4	1.17-1.21	2.2
1.1.3	1.17-1.21	2.2
1.1.2	1.17-1.21	2.0.5 – 2.2
1.1.1	1.17-1.21	2.0.5 – 2.2
1.1.0	1.15-1.21	2.0
1.0	1.15	1.3.2
0.9	1.15	1.3