



Perform volume operations

Astra Trident

NetApp

November 14, 2025

Table of Contents

Perform volume operations	1
Use CSI Topology	1
Step 1: Create a topology-aware backend	2
Step 2: Define StorageClasses that are topology aware	4
Step 3: Create and use a PVC	5
Update backends to include <code>supportedTopologies</code>	8
Find more information	8
Work with snapshots	8
Step 1: Create a <code>VolumeSnapshotClass</code>	8
Step 2: Create a snapshot of an existing PVC	9
Step 3: Create PVCs from VolumeSnapshots	10
Deploying a volume snapshot controller	11
Related links	11
Expand volumes	12
Expand an iSCSI volume	12
Expand an NFS volume	16
Import volumes	19
Drivers that support volume import	19
Why should I import volumes?	20
How does the import work?	20
<code>ontap-nas</code> and <code>ontap-nas-flexgroup</code> imports	21
<code>ontap-san</code> import	22
<code>element</code> import	23
<code>gcp-cvs</code> import	24
<code>azure-netapp-files</code> import	24

Perform volume operations

Learn about the features Astra Trident provides for managing your volumes.

- [Use CSI Topology](#)
- [Work with snapshots](#)
- [Expand volumes](#)
- [Import volumes](#)

Use CSI Topology

Astra Trident can selectively create and attach volumes to nodes present in a Kubernetes cluster by making use of the [CSI Topology feature](#). Using the CSI Topology feature, access to volumes can be limited to a subset of nodes, based on regions and availability zones. Cloud providers today enable Kubernetes administrators to spawn nodes that are zone based. Nodes can be located in different availability zones within a region, or across various regions. To facilitate the provisioning of volumes for workloads in a multi-zone architecture, Astra Trident uses CSI Topology.



Learn more about the CSI Topology feature [here](#).

Kubernetes provides two unique volume binding modes:

- With `VolumeBindingMode` set to `Immediate`, Astra Trident creates the volume without any topology awareness. Volume binding and dynamic provisioning are handled when the PVC is created. This is the default `VolumeBindingMode` and is suited for clusters that do not enforce topology constraints. Persistent Volumes are created without having any dependency on the requesting pod's scheduling requirements.
- With `VolumeBindingMode` set to `WaitForFirstConsumer`, the creation and binding of a Persistent Volume for a PVC is delayed until a pod that uses the PVC is scheduled and created. This way, volumes are created to meet the scheduling constraints that are enforced by topology requirements.



The `WaitForFirstConsumer` binding mode does not require topology labels. This can be used independent of the CSI Topology feature.

What you'll need

To make use of CSI Topology, you need the following:

- A Kubernetes cluster running a [supported Kubernetes version](#)

```

kubectl version
Client Version: version.Info{Major:"1", Minor:"19",
GitVersion:"v1.19.3",
GitCommit:"1e11e4a2108024935ecfcb2912226cedead99df",
GitTreeState:"clean", BuildDate:"2020-10-14T12:50:19Z",
GoVersion:"go1.15.2", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"19",
GitVersion:"v1.19.3",
GitCommit:"1e11e4a2108024935ecfcb2912226cedead99df",
GitTreeState:"clean", BuildDate:"2020-10-14T12:41:49Z",
GoVersion:"go1.15.2", Compiler:"gc", Platform:"linux/amd64"}

```

- Nodes in the cluster should have labels that introduce topology awareness (`topology.kubernetes.io/region` and `topology.kubernetes.io/zone`). These labels **should be present on nodes in the cluster** before Astra Trident is installed for Astra Trident to be topology aware.

```

kubectl get nodes -o=jsonpath='{range .items[*]}{{.metadata.name},\n{.metadata.labels}}{"\n"}{end}' | grep --color "topology.kubernetes.io"
[node1,
{"beta.kubernetes.io/arch":"amd64","beta.kubernetes.io/os":"linux","kubernetes.io/arch":"amd64","kubernetes.io/hostname":"node1","kubernetes.io/os":"linux","node-
role.kubernetes.io/master":"","topology.kubernetes.io/region":"us-
east1","topology.kubernetes.io/zone":"us-east1-a"}]
[node2,
{"beta.kubernetes.io/arch":"amd64","beta.kubernetes.io/os":"linux","kubernetes.io/arch":"amd64","kubernetes.io/hostname":"node2","kubernetes.io/os":"linux","node-
role.kubernetes.io/worker":"","topology.kubernetes.io/region":"us-
east1","topology.kubernetes.io/zone":"us-east1-b"}]
[node3,
{"beta.kubernetes.io/arch":"amd64","beta.kubernetes.io/os":"linux","kubernetes.io/arch":"amd64","kubernetes.io/hostname":"node3","kubernetes.io/os":"linux","node-
role.kubernetes.io/worker":"","topology.kubernetes.io/region":"us-
east1","topology.kubernetes.io/zone":"us-east1-c"}]

```

Step 1: Create a topology-aware backend

Astra Trident storage backends can be designed to selectively provision volumes based on availability zones. Each backend can carry an optional `supportedTopologies` block that represents a list of zones and regions that must be supported. For StorageClasses that make use of such a backend, a volume would only be created if requested by an application that is scheduled in a supported region/zone.

Here is an example backend definition:

YAML

```
---
version: 1
storageDriverName: ontap-san
backendName: san-backend-us-east1
managementLIF: 192.168.27.5
svm: iscsi_svm
username: admin
password: password
supportedTopologies:
- topology.kubernetes.io/region: us-east1
  topology.kubernetes.io/zone: us-east1-a
- topology.kubernetes.io/region: us-east1
  topology.kubernetes.io/zone: us-east1-b
```

JSON

```
{
  "version": 1,
  "storageDriverName": "ontap-san",
  "backendName": "san-backend-us-east1",
  "managementLIF": "192.168.27.5",
  "svm": "iscsi_svm",
  "username": "admin",
  "password": "password",
  "supportedTopologies": [
    {"topology.kubernetes.io/region": "us-east1",
     "topology.kubernetes.io/zone": "us-east1-a"},
    {"topology.kubernetes.io/region": "us-east1",
     "topology.kubernetes.io/zone": "us-east1-b"}
  ]
}
```



`supportedTopologies` is used to provide a list of regions and zones per backend. These regions and zones represent the list of permissible values that can be provided in a `StorageClass`. For `StorageClasses` that contain a subset of the regions and zones provided in a backend, Astra Trident will create a volume on the backend.

You can define `supportedTopologies` per storage pool as well. See the following example:

```
---
version: 1
storageDriverName: ontap-nas
backendName: nas-backend-us-central1
managementLIF: 172.16.238.5
svm: nfs_svm
username: admin
password: password
supportedTopologies:
- topology.kubernetes.io/region: us-central1
  topology.kubernetes.io/zone: us-central1-a
- topology.kubernetes.io/region: us-central1
  topology.kubernetes.io/zone: us-central1-b
storage:
- labels:
    workload: production
    region: Iowa-DC
    zone: Iowa-DC-A
    supportedTopologies:
    - topology.kubernetes.io/region: us-central1
      topology.kubernetes.io/zone: us-central1-a
- labels:
    workload: dev
    region: Iowa-DC
    zone: Iowa-DC-B
    supportedTopologies:
    - topology.kubernetes.io/region: us-central1
      topology.kubernetes.io/zone: us-central1-b
```

In this example, the `region` and `zone` labels stand for the location of the storage pool. `topology.kubernetes.io/region` and `topology.kubernetes.io/zone` dictate where the storage pools can be consumed from.

Step 2: Define StorageClasses that are topology aware

Based on the topology labels that are provided to the nodes in the cluster, StorageClasses can be defined to contain topology information. This will determine the storage pools that serve as candidates for PVC requests made, and the subset of nodes that can make use of the volumes provisioned by Trident.

See the following example:

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: netapp-san-us-east1
  provisioner: csi.trident.netapp.io
  volumeBindingMode: WaitForFirstConsumer
  allowedTopologies:
    - matchLabelExpressions:
    - key: topology.kubernetes.io/zone
      values:
        - us-east1-a
        - us-east1-b
    - key: topology.kubernetes.io/region
      values:
        - us-east1
  parameters:
    fsType: "ext4"

```

In the StorageClass definition provided above, `volumeBindingMode` is set to `WaitForFirstConsumer`. PVCs that are requested with this StorageClass will not be acted upon until they are referenced in a pod. And, `allowedTopologies` provides the zones and region to be used. The `netapp-san-us-east1` StorageClass will create PVCs on the `san-backend-us-east1` backend defined above.

Step 3: Create and use a PVC

With the StorageClass created and mapped to a backend, you can now create PVCs.

See the example spec below:

```

---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-san
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 300Mi
  storageClassName: netapp-san-us-east1

```

Creating a PVC using this manifest would result in the following:

```

kubectl create -f pvc.yaml
persistentvolumeclaim/pvc-san created
kubectl get pvc
NAME      STATUS      VOLUME      CAPACITY      ACCESS MODES      STORAGECLASS
AGE
pvc-san   Pending
2s
2s
kubectl describe pvc
Name:            pvc-san
Namespace:       default
StorageClass:    netapp-san-us-east1
Status:          Pending
Volume:
Labels:          <none>
Annotations:    <none>
Finalizers:     [kubernetes.io/pvc-protection]
Capacity:
Access Modes:
VolumeMode:     Filesystem
Mounted By:    <none>
Events:
  Type  Reason          Age      From          Message
  ----  ----          ----      ----          -----
  Normal  WaitForFirstConsumer  6s      persistentvolume-controller  waiting
for first consumer to be created before binding

```

For Trident to create a volume and bind it to the PVC, use the PVC in a pod. See the following example:

```

apiVersion: v1
kind: Pod
metadata:
  name: app-pod-1
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: topology.kubernetes.io/region
            operator: In
            values:
            - us-east1
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: topology.kubernetes.io/zone
            operator: In
            values:
            - us-east1-a
            - us-east1-b
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  volumes:
  - name: vol1
    persistentVolumeClaim:
      claimName: pvc-san
  containers:
  - name: sec-ctx-demo
    image: busybox
    command: [ "sh", "-c", "sleep 1h" ]
    volumeMounts:
    - name: vol1
      mountPath: /data/demo
    securityContext:
      allowPrivilegeEscalation: false

```

This podSpec instructs Kubernetes to schedule the pod on nodes that are present in the us-east1 region, and choose from any node that is present in the us-east1-a or us-east1-b zones.

See the following output:

```

kubectl get pods -o wide
NAME        READY   STATUS    RESTARTS   AGE   IP           NODE
NOMINATED NODE   READINESS GATES
app-pod-1   1/1     Running   0          19s   192.168.25.131   node2
<none>          <none>
kubectl get pvc -o wide
NAME        STATUS    VOLUME
ACCESS MODES   STORAGECLASS   AGE   VOLUMEMODE
pvc-san     Bound     pvc-ecb1e1a0-840c-463b-8b65-b3d033e2e62b   300Mi
RWO          netapp-san-us-east1   48s   Filesystem

```

Update backends to include `supportedTopologies`

Pre-existing backends can be updated to include a list of `supportedTopologies` using `tridentctl backend update`. This will not affect volumes that have already been provisioned, and will only be used for subsequent PVCs.

Find more information

- [Manage resources for containers](#)
- [nodeSelector](#)
- [Affinity and anti-affinity](#)
- [Taints and Tolerations](#)

Work with snapshots

You can create Kubernetes VolumeSnapshots (volume snapshot) of Persistent Volumes (PVs) to maintain point-in-time copies of Astra Trident volumes. Additionally, you can create a new volume, also known as a *clone*, from an existing volume snapshot. Volume snapshot is supported by `ontap-nas`, `ontap-nas-flexgroup`, `ontap-san`, `ontap-san-economy`, `solidfire-san`, `gcp-cvs`, and `azure-netapp-files` drivers.

Before you begin

You must have an external snapshot controller and Custom Resource Definitions (CRDs). This is the responsibility of the Kubernetes orchestrator (for example: Kubeadm, GKE, OpenShift).

If your Kubernetes distribution does not include the snapshot controller and CRDs, refer to [Deploying a volume snapshot controller](#).



Don't create a snapshot controller if creating on-demand volume snapshots in a GKE environment. GKE uses a built-in, hidden snapshot controller.

Step 1: Create a `VolumeSnapshotClass`

This example creates a volume snapshot class.

```
cat snap-sc.yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
  name: csi-snapclass
driver: csi.trident.netapp.io
deletionPolicy: Delete
```

The `driver` points to Astra Trident's CSI driver. `deletionPolicy` can be `Delete` or `Retain`. When set to `Retain`, the underlying physical snapshot on the storage cluster is retained even when the `VolumeSnapshot` object is deleted.

For more information, refer to [VolumeSnapshotClass](#).

Step 2: Create a snapshot of an existing PVC

This example creates a snapshot of an existing PVC.

```
cat snap.yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: pvc1-snap
spec:
  volumeSnapshotClassName: csi-snapclass
  source:
    persistentVolumeClaimName: pvc1
```

In this example, the snapshot is created for a PVC named `pvc1` and the name of the snapshot is set to `pvc1-snap`.

```
kubectl create -f snap.yaml
volumesnapshot.snapshot.storage.k8s.io/pvc1-snap created

kubectl get volumesnapshots
NAME          AGE
pvc1-snap    50s
```

This created a `VolumeSnapshot` object. A `VolumeSnapshot` is analogous to a PVC and is associated with a `VolumeSnapshotContent` object that represents the actual snapshot.

It is possible to identify the `VolumeSnapshotContent` object for the `pvc1-snap` `VolumeSnapshot` by describing it.

```
kubectl describe volumesnapshots pvcl-snap
Name:          pvcl-snap
Namespace:     default
.
.
.

Spec:
  Snapshot Class Name:  pvcl-snap
  Snapshot Content Name: snapcontent-e8d8a0ca-9826-11e9-9807-525400f3f660
  Source:
    API Group: 
      Kind:      PersistentVolumeClaim
      Name:      pvcl
Status:
  Creation Time: 2019-06-26T15:27:29Z
  Ready To Use:  true
  Restore Size:  3Gi
.
```

The Snapshot Content Name identifies the VolumeSnapshotContent object which serves this snapshot. The Ready To Use parameter indicates that the Snapshot can be used to create a new PVC.

Step 3: Create PVCs from VolumeSnapshots

This example creates a PVC using a snapshot:

```
cat pvc-from-snap.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-from-snap
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: golden
  resources:
    requests:
      storage: 3Gi
  dataSource:
    name: pvcl-snap
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
```

dataSource shows that the PVC must be created using a VolumeSnapshot named `pvc1-snap` as the source of the data. This instructs Astra Trident to create a PVC from the snapshot. After the PVC is created, it can be attached to a pod and used just like any other PVC.

 When deleting a Persistent Volume with associated snapshots, the corresponding Trident volume is updated to a “Deleting state”. For the Astra Trident volume to be deleted, the snapshots of the volume should be removed.

Deploying a volume snapshot controller

If your Kubernetes distribution does not include the snapshot controller and CRDs, you can deploy them as follows.

Steps

1. Create volume snapshot CRDs.

```
cat snapshot-setup.sh
#!/bin/bash
# Create volume snapshot CRDs
kubectl apply -f https://raw.githubusercontent.com/kubernetes-
csi/external-snapshotter/release-
6.1/client/config/crd/snapshot.storage.k8s.io_volumesnapshotclasses.yaml
kubectl apply -f https://raw.githubusercontent.com/kubernetes-
csi/external-snapshotter/release-
6.1/client/config/crd/snapshot.storage.k8s.io_volumesnapshotcontents.yaml
1
kubectl apply -f https://raw.githubusercontent.com/kubernetes-
csi/external-snapshotter/release-
6.1/client/config/crd/snapshot.storage.k8s.io_volumesnapshots.yaml
```

2. Create the snapshot controller in the desired namespace. Edit the YAML manifests below to modify namespace.

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes-
csi/external-snapshotter/release-6.1/deploy/kubernetes/snapshot-
controller/rbac-snapshot-controller.yaml
kubectl apply -f https://raw.githubusercontent.com/kubernetes-
csi/external-snapshotter/release-6.1/deploy/kubernetes/snapshot-
controller/setup-snapshot-controller.yaml
```

Related links

- [Volume snapshots](#)
- [VolumeSnapshotClass](#)

Expand volumes

Astra Trident provides Kubernetes users the ability to expand their volumes after they are created. Find information about the configurations required to expand iSCSI and NFS volumes.

Expand an iSCSI volume

You can expand an iSCSI Persistent Volume (PV) by using the CSI provisioner.



iSCSI volume expansion is supported by the `ontap-san`, `ontap-san-economy`, `solidfire-san` drivers and requires Kubernetes 1.16 and later.

Overview

Expanding an iSCSI PV includes the following steps:

- Editing the StorageClass definition to set the `allowVolumeExpansion` field to `true`.
- Editing the PVC definition and updating the `spec.resources.requests.storage` to reflect the newly desired size, which must be greater than the original size.
- Attaching the PV must be attached to a pod for it to be resized. There are two scenarios when resizing an iSCSI PV:
 - If the PV is attached to a pod, Astra Trident expands the volume on the storage backend, rescans the device, and resizes the filesystem.
 - When attempting to resize an unattached PV, Astra Trident expands the volume on the storage backend. After the PVC is bound to a pod, Trident rescans the device and resizes the filesystem. Kubernetes then updates the PVC size after the expand operation has successfully completed.

The example below shows how expanding iSCSI PVs work.

Step 1: Configure the StorageClass to support volume expansion

```
cat storageclass-ontapsan.yaml
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ontap-san
provisioner: csi.trident.netapp.io
parameters:
  backendType: "ontap-san"
allowVolumeExpansion: True
```

For an already existing StorageClass, edit it to include the `allowVolumeExpansion` parameter.

Step 2: Create a PVC with the StorageClass you created

```
cat pvc-ontapsan.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: san-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: ontap-san
```

Astra Trident creates a Persistent Volume (PV) and associates it with this Persistent Volume Claim (PVC).

```
kubectl get pvc
NAME      STATUS      VOLUME                                     CAPACITY
ACCESS MODES      STORAGECLASS      AGE
san-pvc    Bound      pvc-8a814d62-bd58-4253-b0d1-82f2885db671   1Gi
RWO          ontap-san      8s

kubectl get pv
NAME                                     CAPACITY      ACCESS MODES
RECLAIM POLICY      STATUS      CLAIM      STORAGECLASS      REASON      AGE
pvc-8a814d62-bd58-4253-b0d1-82f2885db671   1Gi          RWO
Delete      Bound      default/san-pvc      ontap-san      10s
```

Step 3: Define a pod that attaches the PVC

In this example, a pod is created that uses the san-pvc.

```
kubectl get pod
NAME          READY   STATUS    RESTARTS   AGE
ubuntu-pod    1/1     Running   0          65s

kubectl describe pvc san-pvc
Name:          san-pvc
Namespace:     default
StorageClass:  ontap-san
Status:        Bound
Volume:        pvc-8a814d62-bd58-4253-b0d1-82f2885db671
Labels:        <none>
Annotations:   pv.kubernetes.io/bind-completed: yes
                pv.kubernetes.io/bound-by-controller: yes
                volume.beta.kubernetes.io/storage-provisioner:
                csi.trident.netapp.io
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      1Gi
Access Modes:  RWO
VolumeMode:    Filesystem
Mounted By:   ubuntu-pod
```

Step 4: Expand the PV

To resize the PV that has been created from 1Gi to 2Gi, edit the PVC definition and update the `spec.resources.requests.storage` to 2Gi.

```

kubectl edit pvc san-pvc
# Please edit the object below. Lines beginning with a '#' will be
ignored,
# and an empty file will abort the edit. If an error occurs while saving
this file will be
# reopened with the relevant failures.
#
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    pv.kubernetes.io/bind-completed: "yes"
    pv.kubernetes.io/bound-by-controller: "yes"
    volume.beta.kubernetes.io/storage-provisioner: csi.trident.netapp.io
  creationTimestamp: "2019-10-10T17:32:29Z"
  finalizers:
  - kubernetes.io/pvc-protection
  name: san-pvc
  namespace: default
  resourceVersion: "16609"
  selfLink: /api/v1/namespaces/default/persistentvolumeclaims/san-pvc
  uid: 8a814d62-bd58-4253-b0d1-82f2885db671
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
...

```

Step 5: Validate the expansion

You can validate the expansion worked correctly by checking the size of the PVC, PV, and the Astra Trident volume:

```

kubectl get pvc san-pvc
NAME      STATUS      VOLUME                                     CAPACITY
ACCESS MODES  STORAGECLASS  AGE
san-pvc    Bound      pvc-8a814d62-bd58-4253-b0d1-82f2885db671  2Gi
RWO          ontap-san   11m

kubectl get pv
NAME                                     CAPACITY  ACCESS MODES
RECLAIM POLICY  STATUS      CLAIM      STORAGECLASS  REASON  AGE
pvc-8a814d62-bd58-4253-b0d1-82f2885db671  2Gi      RWO
Delete        Bound      default/san-pvc  ontap-san           12m

tridentctl get volumes -n trident
+-----+-----+-----+
+-----+-----+-----+
|           NAME          |  SIZE   | STORAGE CLASS | 
PROTOCOL |           BACKEND UUID      | STATE  | MANAGED | 
+-----+-----+-----+
+-----+-----+-----+
| pvc-8a814d62-bd58-4253-b0d1-82f2885db671 | 2.0 GiB | ontap-san      | 
block     | a9b7bfff-0505-4e31-b6c5-59f492e02d33 | online | true      | 
+-----+-----+-----+
+-----+-----+-----+

```

Expand an NFS volume

Astra Trident supports volume expansion for NFS PVs provisioned on `ontap-nas`, `ontap-nas-economy`, `ontap-nas-flexgroup`, `gcp-cvs`, and `azure-netapp-files` backends.

Step 1: Configure the StorageClass to support volume expansion

To resize an NFS PV, the admin first needs to configure the storage class to allow volume expansion by setting the `allowVolumeExpansion` field to `true`:

```

cat storageclass-ontapnas.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ontapnas
provisioner: csi.trident.netapp.io
parameters:
  backendType: ontap-nas
allowVolumeExpansion: true

```

If you have already created a storage class without this option, you can simply edit the existing storage class by using `kubectl edit storageclass` to allow volume expansion.

Step 2: Create a PVC with the StorageClass you created

```
cat pvc-ontapnas.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: ontapnas20mb
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 20Mi
  storageClassName: ontapnas
```

Astra Trident should create a 20MiB NFS PV for this PVC:

```
kubectl get pvc
NAME           STATUS    VOLUME
CAPACITY      ACCESS MODES  STORAGECLASS      AGE
ontapnas20mb  Bound     pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7  20Mi
RWO           ontapnas   9s

kubectl get pv pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7
NAME           CAPACITY      ACCESS MODES
RECLAIM POLICY  STATUS      CLAIM      STORAGECLASS      REASON
AGE
pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7  20Mi      RWO
Delete        Bound     default/ontapnas20mb  ontapnas
2m42s
```

Step 3: Expand the PV

To resize the newly created 20MiB PV to 1GiB, edit the PVC and set `spec.resources.requests.storage` to 1GiB:

```

kubectl edit pvc ontapnas20mb
# Please edit the object below. Lines beginning with a '#' will be
ignored,
# and an empty file will abort the edit. If an error occurs while saving
this file will be
# reopened with the relevant failures.
#
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    pv.kubernetes.io/bind-completed: "yes"
    pv.kubernetes.io/bound-by-controller: "yes"
    volume.beta.kubernetes.io/storage-provisioner: csi.trident.netapp.io
  creationTimestamp: 2018-08-21T18:26:44Z
  finalizers:
  - kubernetes.io/pvc-protection
  name: ontapnas20mb
  namespace: default
  resourceVersion: "1958015"
  selfLink: /api/v1/namespaces/default/persistentvolumeclaims/ontapnas20mb
  uid: c1bd7fa5-a56f-11e8-b8d7-fa163e59eaab
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
...

```

Step 4: Validate the expansion

You can validate the resize worked correctly by checking the size of the PVC, PV, and the Astra Trident volume:

```
kubectl get pvc ontapnas20mb
NAME           STATUS  VOLUME
CAPACITY      ACCESS MODES  STORAGECLASS      AGE
ontapnas20mb  Bound    pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7  1Gi
RWO           ontapnas  4m44s

kubectl get pv pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7
NAME
RECLAIM POLICY  STATUS  CLAIM
AGE
pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7  1Gi      RWO
Delete        Bound    default/ontapnas20mb  ontapnas
5m35s

tridentctl get volume pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7 -n trident
+-----+-----+-----+
+-----+-----+-----+
|           NAME           |  SIZE  | STORAGE CLASS |
PROTOCOL |           BACKEND UUID           | STATE | MANAGED |
+-----+-----+-----+
+-----+-----+-----+
| pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7 | 1.0 GiB | ontapnas      |
file     | c5a6f6a4-b052-423b-80d4-8fb491a14a22 | online | true      |
+-----+-----+-----+
+-----+-----+-----+
```

Import volumes

You can import existing storage volumes as a Kubernetes PV using `tridentctl import`.

Drivers that support volume import

This table depicts the drivers that support importing volumes and the release they were introduced in.

Driver	Release
ontap-nas	19.04
ontap-nas-flexgroup	19.04
solidfire-san	19.04
azure-netapp-files	19.04
gcp-cvs	19.04

Driver	Release
ontap-san	19.04

Why should I import volumes?

There are several use cases for importing a volume into Trident:

- Containerizing an application and reusing its existing data set
- Using a clone of a data set for an ephemeral application
- Rebuilding a failed Kubernetes cluster
- Migrating application data during disaster recovery

How does the import work?

The Persistent Volume Claim (PVC) file is used by the volume import process to create the PVC. At a minimum, the PVC file should include the name, namespace, accessModes, and storageClassName fields as shown in the following example.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my_claim
  namespace: my_namespace
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: my_storage_class
```

The `tridentctl` client is used to import an existing storage volume. Trident imports the volume by persisting volume metadata and creating the PVC and PV.

```
tridentctl import volume <backendName> <volumeName> -f <path-to-pvc-file>
```

To import a storage volume, specify the name of the Astra Trident backend containing the volume, as well as the name that uniquely identifies the volume on the storage (for example: ONTAP FlexVol, Element Volume, CVS Volume path). The storage volume must allow read/write access and be accessible by the specified Astra Trident backend. The `-f` string argument is required and specifies the path to the YAML or JSON PVC file.

When Astra Trident receives the import volume request, the existing volume size is determined and set in the PVC. After the volume is imported by the storage driver, the PV is created with a ClaimRef to the PVC. The reclaim policy is initially set to `retain` in the PV. After Kubernetes successfully binds the PVC and PV, the reclaim policy is updated to match the reclaim policy of the Storage Class. If the reclaim policy of the Storage Class is `delete`, the storage volume will be deleted when the PV is deleted.

When a volume is imported with the `--no-manage` argument, Trident does not perform any additional

operations on the PVC or PV for the lifecycle of the objects. Because Trident ignores PV and PVC events for `--no-manage` objects, the storage volume is not deleted when the PV is deleted. Other operations such as volume clone and volume resize are also ignored. This option is useful if you want to use Kubernetes for containerized workloads but otherwise want to manage the lifecycle of the storage volume outside of Kubernetes.

An annotation is added to the PVC and PV that serves a dual purpose of indicating that the volume was imported and if the PVC and PV are managed. This annotation should not be modified or removed.

Trident 19.07 and later handle the attachment of PVs and mounts the volume as part of importing it. For imports using earlier versions of Astra Trident, there will not be any operations in the data path and the volume import will not verify if the volume can be mounted. If a mistake is made with volume import (for example, the StorageClass is incorrect), you can recover by changing the reclaim policy on the PV to `retain`, deleting the PVC and PV, and retrying the volume import command.

ontap-nas and ontap-nas-flexgroup imports

Each volume created with the `ontap-nas` driver is a FlexVol on the ONTAP cluster. Importing FlexVols with the `ontap-nas` driver works the same. A FlexVol that already exists on an ONTAP cluster can be imported as a `ontap-nas` PVC. Similarly, FlexGroup vols can be imported as `ontap-nas-flexgroup` PVCs.



An ONTAP volume must be of type `rw` to be imported by Trident. If a volume is of type `dp`, it is a SnapMirror destination volume; you should break the mirror relationship before importing the volume into Trident.



The `ontap-nas` driver cannot import and manage qtrees. The `ontap-nas` and `ontap-nas-flexgroup` drivers do not allow duplicate volume names.

For example, to import a volume named `managed_volume` on a backend named `ontap_nas`, use the following command:

```
tridentctl import volume ontap_nas managed_volume -f <path-to-pvc-file>

+-----+-----+-----+
+-----+-----+-----+
|           NAME           |  SIZE   | STORAGE CLASS |
PROTOCOL |           BACKEND UUID           | STATE  | MANAGED |
+-----+-----+-----+
+-----+-----+-----+
| pvc-bf5ad463-afbb-11e9-8d9f-5254004dfdb7 | 1.0 GiB | standard     |
file      | c5a6f6a4-b052-423b-80d4-8fb491a14a22 | online  | true        |
+-----+-----+-----+
+-----+-----+-----+
```

To import a volume named `unmanaged_volume` (on the `ontap_nas` backend), which Trident will not manage, use the following command:

```
tridentctl import volume nas_blog unmanaged_volume -f <path-to-pvc-file>
--no-manage

+-----+-----+-----+
+-----+-----+-----+
|           NAME           | SIZE   | STORAGE CLASS |
PROTOCOL |           BACKEND UUID           | STATE  | MANAGED   |
+-----+-----+-----+
+-----+-----+-----+
| pvc-df07d542-afbc-11e9-8d9f-5254004dfdb7 | 1.0 GiB | standard   |
file     | c5a6f6a4-b052-423b-80d4-8fb491a14a22 | online  | false     |
+-----+-----+-----+
+-----+-----+-----+
```

When using the `--no-manage` argument, Trident does not rename the volume or validate if the volume was mounted. The volume import operation fails if the volume was not mounted manually.



A previously existing bug with importing volumes with custom UnixPermissions has been fixed. You can specify `unixPermissions` in your PVC definition or backend configuration, and instruct Astra Trident to import the volume accordingly.

ontap-san import

Astra Trident can also import ONTAP SAN FlexVols that contain a single LUN. This is consistent with the `ontap-san` driver, which creates a FlexVol for each PVC and a LUN within the FlexVol. You can use the `tridentctl import` command in the same way as in other cases:

- Include the name of the `ontap-san` backend.
- Provide the name of the FlexVol that needs to be imported. Remember, this FlexVol contains only one LUN that must be imported.
- Provide the path of the PVC definition that must be used with the `-f` flag.
- Choose between having the PVC managed or unmanaged. By default, Trident will manage the PVC and rename the FlexVol and LUN on the backend. To import as an unmanaged volume, pass the `--no-manage` flag.



When importing an unmanaged `ontap-san` volume, you should make sure that the LUN in the FlexVol is named `lun0` and is mapped to an igrup with the desired initiators. Astra Trident automatically handles this for a managed import.

Astra Trident will then import the FlexVol and associate it with the PVC definition. Astra Trident also renames the FlexVol to the `pvc-<uuid>` format and the LUN within the FlexVol to `lun0`.



It is recommended to import volumes that do not have existing active connections. If you are looking to import an actively used volume, clone the volume first and then do the import.

Example

To import the `ontap-san`-managed FlexVol that is present on the `ontap_san_default` backend, run the `tridentctl import` command as:

```
tridentctl import volume ontapsan_san_default ontap-san-managed -f pvc-basic-import.yaml -n trident -d

+-----+-----+-----+
+-----+-----+-----+
|           NAME           |  SIZE  | STORAGE CLASS  |
| PROTOCOL | BACKEND UUID      | STATE  | MANAGED  |
+-----+-----+-----+
+-----+-----+-----+
| pvc-d6ee4f54-4e40-4454-92fd-d00fc228d74a | 20 MiB | basic          |
| block    | cd394786-ddd5-4470-adc3-10c5ce4ca757 | online | true       |
+-----+-----+-----+
+-----+-----+-----+
```



An ONTAP volume must be of type `rw` to be imported by Astra Trident. If a volume is of type `dp`, it is a SnapMirror destination volume; you should break the mirror relationship before importing the volume into Astra Trident.

element `import`

You can import NetApp Element software/NetApp HCI volumes to your Kubernetes cluster with Trident. You need the name of your Astra Trident backend, and the unique name of the volume and the PVC file as the arguments for the `tridentctl import` command.

```
tridentctl import volume element_default element-managed -f pvc-basic-import.yaml -n trident -d

+-----+-----+-----+
+-----+-----+-----+
|           NAME           |  SIZE  | STORAGE CLASS  |
| PROTOCOL | BACKEND UUID      | STATE  | MANAGED  |
+-----+-----+-----+
+-----+-----+-----+
| pvc-970ce1ca-2096-4ecd-8545-ac7edc24a8fe | 10 GiB | basic-element |
| block    | d3ba047a-ea0b-43f9-9c42-e38e58301c49 | online | true       |
+-----+-----+-----+
+-----+-----+-----+
```



The Element driver supports duplicate volume names. If there are duplicate volume names, Trident's volume import process returns an error. As a workaround, clone the volume and provide a unique volume name. Then import the cloned volume.

gcp-cvs import



To import a volume backed by the NetApp Cloud Volumes Service in GCP, identify the volume by its volume path instead of its name.

To import an `gcp-cvs` volume on the backend called `gcpcvs_YEppr` with the volume path of `adroit-jolly-swift`, use the following command:

```
tridentctl import volume gcpcvs_YEppr adroit-jolly-swift -f <path-to-pvc-file> -n trident

+-----+-----+-----+
+-----+-----+-----+-----+
|           NAME           |  SIZE  | STORAGE CLASS | 
PROTOCOL |           BACKEND UUID           | STATE | MANAGED | 
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| pvc-a46ccab7-44aa-4433-94b1-e47fc8c0fa55 | 93 GiB | gcp-storage | file
| e1a6e65b-299e-4568-ad05-4f0a105c888f | online | true     |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
```



The volume path is the portion of the volume's export path after the `:`. For example, if the export path is `10.0.0.1:/adroit-jolly-swift`, the volume path is `adroit-jolly-swift`.

azure-netapp-files import

To import an `azure-netapp-files` volume on the backend called `azurenatappfiles_40517` with the volume path `importvol1`, run the following command:

```
tridentctl import volume azurenetaappfiles_40517 importvol1 -f <path-to-pvc-file> -n trident
```

PROTOCOL	NAME	SIZE	STORAGE CLASS	STATE	MANAGED
	BACKEND UUID				
	pvc-0ee95d60-fd5c-448d-b505-b72901b3a4ab	100 GiB	anf-storage		
file	1c01274f-d94b-44a3-98a3-04c953c9a51e			online	true



The volume path for the ANF volume is present in the mount path after the `:`. For example, if the mount path is `10.0.0.2:/importvol1`, the volume path is `importvol1`.

Copyright information

Copyright © 2025 NetApp, Inc. All Rights Reserved. Printed in the U.S. No part of this document covered by copyright may be reproduced in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an electronic retrieval system—without prior written permission of the copyright owner.

Software derived from copyrighted NetApp material is subject to the following license and disclaimer:

THIS SOFTWARE IS PROVIDED BY NETAPP “AS IS” AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WHICH ARE HEREBY DISCLAIMED. IN NO EVENT SHALL NETAPP BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

NetApp reserves the right to change any products described herein at any time, and without notice. NetApp assumes no responsibility or liability arising from the use of products described herein, except as expressly agreed to in writing by NetApp. The use or purchase of this product does not convey a license under any patent rights, trademark rights, or any other intellectual property rights of NetApp.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

LIMITED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (b)(3) of the Rights in Technical Data -Noncommercial Items at DFARS 252.227-7013 (FEB 2014) and FAR 52.227-19 (DEC 2007).

Data contained herein pertains to a commercial product and/or commercial service (as defined in FAR 2.101) and is proprietary to NetApp, Inc. All NetApp technical data and computer software provided under this Agreement is commercial in nature and developed solely at private expense. The U.S. Government has a non-exclusive, non-transferrable, nonsublicensable, worldwide, limited irrevocable license to use the Data only in connection with and in support of the U.S. Government contract under which the Data was delivered. Except as provided herein, the Data may not be used, disclosed, reproduced, modified, performed, or displayed without the prior written approval of NetApp, Inc. United States Government license rights for the Department of Defense are limited to those rights identified in DFARS clause 252.227-7015(b) (FEB 2014).

Trademark information

NETAPP, the NETAPP logo, and the marks listed at <http://www.netapp.com/TM> are trademarks of NetApp, Inc. Other company and product names may be trademarks of their respective owners.