Provision and manage volumes
Astra Trident
NetApp
March 05, 2024

This PDF was generated from https://docs.netapp.com/us-en/trident/trident-use/vol-provision.html on March 05, 2024. Always check docs.netapp.com for the latest.
# Table of Contents

Provision and manage volumes .......................... 1
  Provision a volume .................................. 1
  Expand volumes ....................................... 5
  Import volumes ....................................... 12
Share an NFS volume across namespaces ................. 20
Use CSI Topology ....................................... 24
Work with snapshots .................................... 32
Provision and manage volumes

Provision a volume

Create a PersistentVolume (PV) and a PersistentVolumeClaim (PVC) that uses the configured Kubernetes StorageClass to request access to the PV. You can then mount the PV to a pod.

Overview

A PersistentVolume (PV) is a physical storage resource provisioned by the cluster administrator on a Kubernetes cluster. The PersistentVolumeClaim (PVC) is a request for access to the PersistentVolume on the cluster.

The PVC can be configured to request storage of a certain size or access mode. Using the associated StorageClass, the cluster administrator can control more than PersistentVolume size and access mode—such as performance or service level.

After you create the PV and PVC, you can mount the volume in a pod.

Sample manifests

PersistentVolume sample manifest

This sample manifest shows a basic PV of 10Gi that is associated with StorageClass basic-csi.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-storage
  labels:
    type: local
spec:
  storageClassName: basic-csi
  capacity:
    storage: 10Gi
  accessModes:
  - ReadWriteOnce
hostPath:
  path: "/my/host/path"
```
PersistentVolumeClaim sample manifests

These examples show basic PVC configuration options.

**PVC with RWO access**
This example shows a basic PVC with RWO access that is associated with a StorageClass named `basic-csi`.

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-storage
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: basic-csi
```

**PVC with NVMe/TCP**
This example shows a basic PVC for NVMe/TCP with RWO access that is associated with a StorageClass named `protection-gold`.

```yaml
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-san-nvme
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 300Mi
  storageClassName: protection-gold
```
Pod manifest samples

These examples show basic configurations to attach the PVC to a pod.

Basic configuration

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: pv-pod
spec:
  volumes:
  - name: pv-storage
    persistentVolumeClaim:
      claimName: basic
  containers:
  - name: pv-container
    image: nginx
    ports:
      - containerPort: 80
        name: "http-server"
    volumeMounts:
      - mountPath: "/my/mount/path"
        name: pv-storage
```
Basic NVMe/TCP configuration

---
apiVersion: v1
type: Pod
metadata:
  creationTimestamp: null
labels:
  run: nginx
  name: nginx
spec:
  containers:
  - image: nginx
  name: nginx
  resources: {}
  volumeMounts:
  - mountPath: "/usr/share/nginx/html"
    name: task-pv-storage
dnsPolicy: ClusterFirst
restartPolicy: Always
volumes:
  - name: task-pv-storage
    persistentVolumeClaim:
      claimName: pvc-san-nvme

---

Create the PV and PVC

Steps
1. Create the PV.

   kubectl create -f pv.yaml

2. Verify the PV status.

   kubectl get pv

<table>
<thead>
<tr>
<th>NAME</th>
<th>CAPACITY</th>
<th>ACCESS MODES</th>
<th>RECLAIM POLICY</th>
<th>STATUS</th>
<th>CLAIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>pv-storage</td>
<td>4Gi</td>
<td>RWO</td>
<td>Retain</td>
<td>Available</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7s</td>
<td></td>
</tr>
</tbody>
</table>

3. Create the PVC.
kubectl create -f pvc.yaml

4. Verify the PVC status.

```shell
kubectl get pvc
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>STATUS</th>
<th>VOLUME</th>
<th>CAPACITY</th>
<th>ACCESS MODES</th>
<th>STORAGECLASS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvc-storage</td>
<td>Bound</td>
<td>pv-name</td>
<td>2Gi</td>
<td>RWO</td>
<td></td>
<td>5m</td>
</tr>
</tbody>
</table>

5. Mount the volume in a pod.

```shell
kubectl create -f pv-pod.yaml
```

ℹ️ You can monitor the progress using `kubectl get pod --watch`.

6. Verify that the volume is mounted on `/my/mount/path`.

```shell
kubectl exec -it task-pv-pod -- df -h /my/mount/path
```

7. You can now delete the Pod. The Pod application will no longer exist, but the volume will remain.

```shell
kubectl delete pod task-pv-pod
```

Refer to [Kubernetes and Trident objects](#) for details on how storage classes interact with the `PersistentVolumeClaim` and parameters for controlling how Astra Trident provisions volumes.

## 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.

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

**Step 1: Configure the StorageClass to support volume expansion**

Edit the StorageClass definition to set the `allowVolumeExpansion` field to `true`. 
For an already existing StorageClass, edit it to include the allowVolumeExpansion parameter.

Step 2: Create a PVC with the StorageClass you created

Edit the PVC definition and update the spec.resources.requests.storage to reflect the newly desired size, which must be greater than the original size.

Astra Trident creates a Persistent Volume (PV) and associates it with this Persistent Volume Claim (PVC).
kubectl get pvc
<table>
<thead>
<tr>
<th>NAME</th>
<th>STATUS</th>
<th>VOLUME</th>
<th>CAPACITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>san-pvc</td>
<td>Bound</td>
<td>pvc-8a814d62-bd58-4253-b0d1-82f2885db671</td>
<td>1Gi</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RWO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ontap-san</td>
</tr>
</tbody>
</table>

kubectl get pv
<table>
<thead>
<tr>
<th>NAME</th>
<th>CAPACITY</th>
<th>ACCESS MODES</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvc-8a814d62-bd58-4253-b0d1-82f2885db671</td>
<td>1Gi</td>
<td>RWO</td>
</tr>
<tr>
<td>Delete</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Delete: Bound default/san-pvc ontap-san 10s

Step 3: Define a pod that attaches the PVC

Attach the PV 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.

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

kubectl get pod
<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ubuntu-pod</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>65s</td>
</tr>
</tbody>
</table>

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:
<table>
<thead>
<tr>
<th>NAME</th>
<th>STATUS</th>
<th>VOLUME</th>
<th>CAPACITY</th>
<th>ACCESS MODES</th>
<th>STORAGECLASS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>san-pvc</td>
<td>Bound</td>
<td>pvc-8a814d62-bd58-4253-b0d1-82f2885db671</td>
<td>2Gi</td>
<td>RWO</td>
<td>ontap-san</td>
<td>11m</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>CAPACITY</th>
<th>ACCESS MODES</th>
<th>RECLAIM POLICY</th>
<th>STATUS</th>
<th>CLAIM</th>
<th>STORAGECLASS</th>
<th>REASON</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvc-8a814d62-bd58-4253-b0d1-82f2885db671</td>
<td>2Gi</td>
<td>RWO</td>
<td>Delete</td>
<td>Bound</td>
<td>default/san-pvc</td>
<td>ontap-san</td>
<td></td>
<td>12m</td>
</tr>
</tbody>
</table>

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

```yaml
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
ontapnas20mb                                20Mi        RWO

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:
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                   CAPACITY   ACCESS MODES
RECLAIM POLICY STATUS CLAIM                  STORAGECLASS    REASON AGEl
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`.

Overview and considerations

You might import a volume into Astra Trident to:

- Containerize an application and reuse its existing data set
- Use a clone of a data set for an ephemeral application
- Rebuild a failed Kubernetes cluster
- Migrate application data during disaster recovery

Considerations

Before importing a volume, review the following considerations.

- Astra Trident can import RW (read-write) type ONTAP volumes only. DP (data protection) type volumes are SnapMirror destination volumes. You should break the mirror relationship before importing the volume into
Astra Trident.

- We suggest importing volumes without active connections. To import an actively-used volume, clone the volume and then perform the import.

  This is especially important for block volumes as Kubernetes would be unaware of the previous connection and could easily attach an active volume to a pod. This can result in data corruption.

- Though StorageClass must be specified on a PVC, Astra Trident does not use this parameter during import. Storage classes are used during volume creation to select from available pools based on storage characteristics. Because the volume already exists, no pool selection is required during import. Therefore, the import will not fail even if the volume exists on a backend or pool that does not match the storage class specified in the PVC.

- 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.

- By default, Astra Trident manages the PVC and renames the FlexVol and LUN on the backend. You can pass the --no-manage flag to import an unmanaged volume. If you use --no-manage, Astra Trident does not perform any additional operations on the PVC or PV for the lifecycle of the objects. The storage volume is not deleted when the PV is deleted and 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.

**Import a volume**

You can use tridentctl import to import a volume.

**Steps**

1. Create the Persistent Volume Claim (PVC) file (for example, pvc.yaml) that will be used to create the PVC. The PVC file should include name, namespace, accessModes, and storageClassName. Optionally, you can specify unixPermissions in your PVC definition.

   The following is an example of a minimum specification:
Don't include additional parameters such as PV name or volume size. This can cause the import command to fail.

2. Use the `tridentctl import` command to specify the name of the Astra Trident backend containing the volume and the name that uniquely identifies the volume on the storage (for example: ONTAP FlexVol, Element Volume, Cloud Volumes Service path). The `-f` argument is required to specify the path to the PVC file.

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

Examples

Review the following volume import examples for supported drivers.

**ONTAP NAS and ONTAP NAS FlexGroup**

Astra Trident supports volume import using the `ontap-nas` and `ontap-nas-flexgroup` drivers.

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

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.

**ONTAP NAS examples**

The following show an example of a managed volume and an unmanaged volume import.
**Managed volume**

The following example imports a volume named `managed_volume` on a backend named `ontap_nas`:

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

<table>
<thead>
<tr>
<th>NAME</th>
<th>SIZE</th>
<th>STORAGE CLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROTOCOL</td>
<td>BACKEND UUID</td>
<td>STATE</td>
</tr>
<tr>
<td>file</td>
<td>c5a6f6a4-b052-423b-80d4-8fb491a14a22</td>
<td>online</td>
</tr>
</tbody>
</table>

**Unmanaged volume**

When using the `--no-manage` argument, Astra Trident does not rename the volume.

The following example imports `unmanaged_volume` on the `ontap_nas` backend:

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

<table>
<thead>
<tr>
<th>NAME</th>
<th>SIZE</th>
<th>STORAGE CLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROTOCOL</td>
<td>BACKEND UUID</td>
<td>STATE</td>
</tr>
<tr>
<td>file</td>
<td>c5a6f6a4-b052-423b-80d4-8fb491a14a22</td>
<td>online</td>
</tr>
</tbody>
</table>

**ONTAP SAN**

Astra Trident supports volume import using the `ontap-san` driver. Volume import is not supported using the `ontap-san-economy` driver.

Astra Trident can 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. Astra Trident imports the FlexVol and associates it with the PVC definition.
ONTAP SAN examples

The following show an example of a managed volume and an unmanaged volume import.
Managed volume

For managed volumes, Astra Trident renames the FlexVol to the `pvc-<uuid>` format and the LUN within the FlexVol to `lun0`.

The following example imports the `ontap-san-managed` FlexVol that is present on the `ontap_san_default` backend:

```
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    |
+------------------------------------------+--------+---------------+
```

Unmanaged volume

The following example imports `unmanaged_example_volume` on the `ontap_san` backend:

```
tridentctl import volume -n trident san_blog unmanaged_example_volume -f pvc-import.yaml --no-manage
```

```
+------------------------------------------+---------+---------------+
|                   NAME                   |  SIZE   | STORAGE CLASS |
| PROTOCOL |             BACKEND UUID             | STATE  | MANAGED |
+------------------------------------------+---------+---------------+
| pvc-1fc999c9-ce8c-459c-82e4-ed4380a4b228 | 1.0 GiB | san-blog      |
| block    | e3275890-7d80-4af6-90cc-c7a0759f555a | online | false   |
+------------------------------------------+---------+---------------+
```
If you have LUNS mapped to igroups that share an IQN with a Kubernetes node IQN, as shown in the following example, you will receive the error: LUN already mapped to initiator(s) in this group. You will need to remove the initiator or unmap the LUN to import the volume.

```
+------------------------------------------+--------+---------------+
| Vserver  | Igroup             | Protocol | OS Type | Initiators                             |
|----------+--------------------+----------+---------+---------------------------------------|
| svm0     | k8s-nodename.example.com-fe5d36f2-cded.4f38-9eb0-c7719fc2f9f3 | iscsi    | linux   | ign.1994-05.com.redhat:4c2e1cf35e0   |
| svm0     | unmanaged-example-igroup | mixed   | linux   | ign.1994-05.com.redhat:4c2e1cf35e0   |
+------------------------------------------+--------+---------------+
```

### Element

Astra Trident supports NetApp Element software and NetApp HCI volume import using the `solidfire-san` driver.

The Element driver supports duplicate volume names. However, Astra Trident returns an error if there are duplicate volume names. As a workaround, clone the volume, provide a unique volume name, and import the cloned volume.

### Element example

The following example imports an `element-managed` volume on backend `element_default`.

```
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 | online | true   |
+---------------------------------------------------------------------+--------+---------------+
```

### Google Cloud Platform

Astra Trident supports volume import using the `gcp-cvs` driver.
To import a volume backed by the NetApp Cloud Volumes Service in Google Cloud Platform, identify the volume by its volume path. 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.

**Google Cloud Platform example**
The following example imports a `gcp-cvs` volume on backend `gcpcvs_YEppr` with the volume path of adroit-jolly-swift.

```
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    |
+-------------------------------------------------+--------+---------------+

**Azure NetApp Files**

Astra Trident supports volume import using the `azure-netapp-files` driver.

To import an Azure NetApp Files volume, identify the volume by its volume path. The volume path is the portion of the volume’s export path after the :/. For example, if the mount path is 10.0.0.2:/importvol1, the volume path is importvol1.

**Azure NetApp Files example**
The following example imports an `azure-netapp-files` volume on backend `azurenetappfiles_40517` with the volume path `importvol1`.

```
```
tridentctl import volume azurenetappfiles_40517 importvol1 -f <path-to-pvc-file> -n trident

<table>
<thead>
<tr>
<th>NAME</th>
<th>SIZE</th>
<th>STORAGE CLASS</th>
<th>PROTOCOL</th>
<th>BACKEND UUID</th>
<th>STATE</th>
<th>MANAGED</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvc-0ee95d60-fd5c-448d-b505-b72901b3a4ab</td>
<td>100 GiB</td>
<td>anf-storage</td>
<td>file</td>
<td>1c01274f-d94b-44a3-98a3-04c953c9a51e</td>
<td>online</td>
<td>true</td>
</tr>
</tbody>
</table>

Share an NFS volume across namespaces

Using Astra Trident, you can create a volume in a primary namespace and share it in one or more secondary namespaces.

Features

The Astra TridentVolumeReference CR allows you to securely share ReadWriteMany (RWX) NFS volumes across one or more Kubernetes namespaces. This Kubernetes-native solution has the following benefits:

- Multiple levels of access control to ensure security
- Works with all Trident NFS volume drivers
- No reliance on tridentctl or any other non-native Kubernetes feature

This diagram illustrates NFS volume sharing across two Kubernetes namespaces.
Quick start
You can set up NFS volume sharing in just a few steps.

1. Configure source PVC to share the volume
The source namespace owner grants permission to access the data in the source PVC.

2. Grant permission to create a CR in the destination namespace
The cluster administrator grants permission to the owner of the destination namespace to create the TridentVolumeReference CR.

3. Create TridentVolumeReference in the destination namespace
The owner of the destination namespace creates the TridentVolumeReference CR to refer to the source PVC.

4. Create the subordinate PVC in the destination namespace
The owner of the destination namespace creates the subordinate PVC to use the data source from the source PVC.
Configure the source and destination namespaces

To ensure security, cross namespace sharing requires collaboration and action by the source namespace owner, cluster administrator, and destination namespace owner. The user role is designated in each step.

Steps

1. **Source namespace owner:** Create the PVC (pvc1) in the source namespace that grants permission to share with the destination namespace (namespace2) using the shareToNamespace annotation.

   ```yaml
   apiVersion: v1
   kind: PersistentVolumeClaim
   metadata:
     name: pvc1
     namespace: namespace1
     annotations:
       trident.netapp.io/shareToNamespace: namespace2
   spec:
     accessModes:
       - ReadWriteMany
     storageClassName: trident-csi
     resources:
       requests:
         storage: 100Gi
   ```

   Astra Trident creates the PV and its backend NFS storage volume.

   - You can share the PVC to multiple namespaces using a comma-delimited list. For example, trident.netapp.io/shareToNamespace: namespace2,namespace3,namespace4.

   - You can share to all namespaces using *. For example, trident.netapp.io/shareToNamespace: *

   - You can update the PVC to include the shareToNamespace annotation at any time.

2. **Cluster admin:** Create the custom role and kubeconfig to grant permission to the destination namespace owner to create the TridentVolumeReference CR in the destination namespace.

3. **Destination namespace owner:** Create a TridentVolumeReference CR in the destination namespace that refers to the source namespace pvc1.
apiVersion: trident.netapp.io/v1  
kind: TridentVolumeReference  
metadata:  
  name: my-first-tvr  
  namespace: namespace2  
spec:  
  pvcName: pvc1  
  pvcNamespace: namespace1

4. **Destination namespace owner:** Create a PVC (pvc2) in destination namespace (namespace2) using the `shareFromPVC` annotation to designate the source PVC.

```yaml  
kind: PersistentVolumeClaim  
apiVersion: v1  
metadata:  
  annotations:  
    trident.netapp.io/shareFromPVC: namespace1/pvc1  
  name: pvc2  
  namespace: namespace2  
spec:  
  accessModes:  
    - ReadWriteMany  
  storageClassName: trident-csi  
  resources:  
    requests:  
      storage: 100Gi
```

⚠️ The size of the destination PVC must be less than or equal than the source PVC.

**Results**

Astra Trident reads the `shareFromPVC` annotation on the destination PVC and creates the destination PV as a subordinate volume with no storage resource of its own that points to the source PV and shares the source PV storage resource. The destination PVC and PV appear bound as normal.

**Delete a shared volume**

You can delete a volume that is shared across multiple namespaces. Astra Trident will remove access to the volume on the source namespace and maintain access for other namespaces that share the volume. When all namespaces that reference the volume are removed, Astra Trident deletes the volume.

**Use `tridentctl get` to query subordinate volumes**

Using the `tridentctl` utility, you can run the `get` command to get subordinate volumes. For more information, refer to `tridentctl commands and options`. 

---

23
Usage:
tridentctl get [option]

Flags:

• `-h, --help`: Help for volumes.

• `--parentOfSubordinate string`: Limit query to subordinate source volume.

• `--subordinateOf string`: Limit query to subordinates of volume.

Limitations

• Astra Trident cannot prevent destination namespaces from writing to the shared volume. You should use file locking or other processes to prevent overwriting shared volume data.

• You cannot revoke access to the source PVC by removing the `shareToNamespace` or `shareFromNamespace` annotations or deleting the `TridentVolumeReference` CR. To revoke access, you must delete the subordinate PVC.

• Snapshots, clones, and mirroring are not possible on subordinate volumes.

For more information

To learn more about cross-namespace volume access:

• Visit [Sharing volumes between namespaces: Say hello to cross-namespace volume access.](#)

• Watch the demo on [NetAppTV.](#)

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.](#)

Overview

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**

  ```bash
  kubectl version
  Client Version: version.Info{Major:"1", Minor:"19",
  GitVersion:"v1.19.3",
  GitCommit:"1e11e4a2108024935ecfcb2912226cedae69df",
  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:"1e11e4a2108024935ecfcb2912226cedae69df",
  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},{.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

```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

```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:
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:
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:

```yaml
---
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                                      netapp-san-us-east1
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
class: 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:
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

Kubernetes volume snapshots of Persistent Volumes (PVs) enable point-in-time copies of volumes. You can create a snapshot of a volume created using Astra Trident, import a snapshot created outside of Astra Trident, create a new volume from an existing snapshot, and recover volume data from snapshots.

Overview

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) to work with snapshots. 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 Deploy 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.
Create a volume snapshot

Steps

1. Create a `VolumeSnapshotClass`. For more information, refer to `VolumeSnapshotClass`.
   - The `driver` points to the Astra Trident 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.

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

2. Create a snapshot of an existing PVC.

   Examples
   - This example creates a snapshot of an existing PVC.

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

   - This example creates a volume snapshot object for a PVC named `pvc1` and the name of the snapshot is set to `pvc1-snap`. A `VolumeSnapshot` is analogous to a PVC and is associated with a `VolumeSnapshotContent` object that represents the actual snapshot.

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

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

   - You can identify the `VolumeSnapshotContent` object for the `pvc1-snap` `VolumeSnapshot` by
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.

```
kubectl describe volumesnapshots pvc1-snap
Name:         pvc1-snap
Namespace:    default
.
.
Spec:
  Snapshot Class Name:    pvc1-snap
  Snapshot Content Name:  snapcontent-e8d8a0ca-9826-11e9-9807-525400f3f660
  Source:
    Kind:       PersistentVolumeClaim
    Name:       pvc1
Status:
  Ready To Use:   true
  Restore Size:   3Gi
```

Create a PVC from a volume snapshot

You can use dataSource to create a PVC using a VolumeSnapshot named `<pvc-name>` as the source of the data. After the PVC is created, it can be attached to a pod and used just like any other PVC.

The PVC will be created in the same backend as the source volume. Refer to KB: Creating a PVC from a Trident PVC Snapshot cannot be created in an alternate backend.

The following example creates the PVC using `pvc1-snap` as the data source.
Import a volume snapshot

Astra Trident supports the Kubernetes pre-provisioned snapshot process to enable the cluster administrator to create a VolumeSnapshotContent object and import snapshots created outside of Astra Trident.

Before you begin
Astra Trident must have created or imported the snapshot’s parent volume.

Steps
1. **Cluster admin:** Create a VolumeSnapshotContent object that references the backend snapshot. This initiates the snapshot workflow in Astra Trident.
   - Specify the name of the backend snapshot in annotations as trident.netapp.io/internalSnapshotName: "backend-snapshot-name".
   - Specify `<name-of-parent-volume-in-trident>/<volume-snapshot-content-name>` in snapshotHandle. This is the only information provided to Astra Trident by the external snapshotter in the `ListSnapshots` call.

   The `<volumeSnapshotContentName>` cannot always match the backend snapshot name due to CR naming constraints.

Example
The following example creates a VolumeSnapshotContent object that references backend snapshot snap-01.
2. **Cluster admin:** Create the `VolumeSnapshot` CR that references the `VolumeSnapshotContent` object. This requests access to use the `VolumeSnapshot` in a given namespace.

**Example**

The following example creates a `VolumeSnapshot` CR named `import-snap` that references the `VolumeSnapshotContent` named `import-snap-content`.

```yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: import-snap
spec:
  # volumeSnapshotClassName: csi-snapclass (not required for pre-provisioned or imported snapshots)
  source:
    volumeSnapshotContentName: import-snap-content
```

3. **Internal processing (no action required):** The external snapshotter recognizes the newly created `VolumeSnapshotContent` and runs the `ListSnapshots` call. Astra Trident creates the `TridentSnapshot`.

   - The external snapshotter sets the `VolumeSnapshotContent` to `readyToUse` and the `VolumeSnapshot` to `true`.
   - Trident returns `readyToUse=true`.

4. **Any user:** Create a `PersistentVolumeClaim` to reference the new `VolumeSnapshot`, where the `spec.dataSource` (or `spec.dataSourceRef`) name is the `VolumeSnapshot` name.

**Example**

The following example creates a PVC referencing the `VolumeSnapshot` named `import-snap`.

```yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotContent
metadata:
  name: import-snap-content
  annotations:
    trident.netapp.io/internalSnapshotName: "snap-01"  # This is the name of the snapshot on the backend
spec:
  deletionPolicy: Retain
  driver: csi.trident.netapp.io
  source:
    snapshotHandle: pvc-f71223b5-23b9-4235-bbfe-e269ac7b84b0/import-snap-content # <import PV name or source PV name>/<volume-snapshot-content-name>
```
Recover volume data using snapshots

The snapshot directory is hidden by default to facilitate maximum compatibility of volumes provisioned using the `ontap-nas` and `ontap-nas-economy` drivers. Enable the `.snapshot` directory to recover data from snapshots directly.

Use the volume snapshot restore ONTAP CLI to restore a volume to a state recorded in a prior snapshot.

```
cluster1::*> volume snapshot restore -vserver vs0 -volume vol3 -snapshot vol3_snap_archive
```

When you restore a snapshot copy, the existing volume configuration is overwritten. Changes made to volume data after the snapshot copy was created are lost.

Delete a PV with associated snapshots

When deleting a Persistent Volume with associated snapshots, the corresponding Trident volume is updated to a “Deleting state”. Remove the volume snapshots to delete the Astra Trident volume.

Deploy 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
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.


If necessary, open deploy/kubernetes/snapshot-controller/rbac-snapshot-controller.yaml and update namespace to your namespace.

Related links

- Volume snapshots
- VolumeSnapshotClass
Copyright information

Copyright © 2024 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.