Reference

Trident

NetApp
June 30, 2026

This PDF was generated from https://docs.netapp.com/us-en/trident-2602/trident-reference/ports.html on June 30, 2026. Always check docs.netapp.com for the latest.

Table of Contents

Reference

Trident ports

Learn more about the ports that Trident uses for communication.

Overview

Trident uses various ports for communication inside Kubernetes clusters and with storage backends. The following is a summary of key ports, their purposes, and security considerations.

- **Outbound focus**: Kubernetes nodes (controller and worker) primarily initiate traffic to storage LIFs/IPs, so iptables rules should allow outbound from node IPs to specific storage IPs on these ports. Avoid broad "any-to-any" rules.
- **Inbound restrictions**: Limit internal Trident ports to cluster-internal traffic (for example, using CNI like Calico). No unnecessary inbound exposure on host firewalls.
- **Protocol security**:
  - Use TCP where possible (more reliable).
  - Enable CHAP/IPsec for iSCSI if sensitive; TLS/HTTPS for management (port 443/8443).
  - For NFSv4 (default in Trident), prune UDP/older NFSv3 ports (for example, 4045-4049) if not needed.
  - Restrict to trusted subnets; monitor with tools like Prometheus (optional port 8001).

Ports for controller nodes

These ports are primarily for Trident operator (backend management). All internal ports are pod-level; allow on nodes only if host firewall interferes with CNI.

Port/Protocol	Direction	Purpose	Driver/Protocol	Security Notes
TCP 8000	Inbound/Outbound (cluster-internal)	Trident REST server (operator-controller comms)	All	Restrict to pod CIDRs; no external exposure.
TCP 8443	Inbound/Outbound (cluster-internal)	Backchannel HTTPS (secure internal API)	All	TLS-encrypted; limit to Kubernetes service mesh if used.
TCP 8001	Inbound (cluster-internal, optional)	Prometheus metrics	All	Expose only to monitoring tools (for example, using RBAC); disable if unused.
TCP 443	Outbound	HTTPS to ONTAP SVM/cluster mgmt LIF	ONTAP (all), ANF	Require TLS cert validation; restrict to mgmt LIF IPs only.
TCP 8443	Outbound	HTTPS to E-Series Web Services Proxy	E-Series (iSCSI)	Default REST API; use certs; configurable in backend YAML.

Ports for worker nodes

These ports are for CSI node daemonsets and pod mounts. Data ports are outbound to storage data LIFs; include NFSv3 extras if using NFSv3 (optional for NFSv4).

Port/Protocol	Direction	Purpose	Driver/Protocol	Security Notes
TCP 17546	Inbound (local to pod)	CSI node liveness/readiness probes	All	Configurable (--probe-port); ensure no host conflicts; local-only.
TCP 8000	Inbound/Outbound (cluster-internal)	Trident REST server	All	As above; pod-internal.
TCP 8443	Inbound/Outbound (cluster-internal)	Backchannel HTTPS	All	As above.
TCP 8001	Inbound (cluster-internal, optional)	Prometheus metrics	All	As above.
TCP 443	Outbound	HTTPS to ONTAP SVM/cluster mgmt LIF	ONTAP (all), ANF	As above; used for discovery.
TCP 8443	Outbound	HTTPS to E-Series Web Services Proxy	E-Series (iSCSI)	As above.
TCP/UDP 111	Outbound	RPCBIND/portmapper	ONTAP-NAS (NFSv3/v4), ANF (NFS)	Required for v3; optional for v4 (firewall offload); restrict if using NFSv4-only.
TCP/UDP 2049	Outbound	NFS daemon	ONTAP-NAS (NFSv3/v4), ANF (NFS)	Core data; well-known; use TCP for reliability.
TCP/UDP 635	Outbound	Mount daemon	ONTAP-NAS (NFSv3/v4), ANF (NFS)	Mounting; bidirectional callbacks possible (allow inbound ephemeral if needed).
UDP 4045	Outbound	NFS lock manager (nlockmgr)	ONTAP-NAS (NFSv3)	File locking; skip for v4 (pNFS handles); UDP-only.
UDP 4046	Outbound	NFS status monitor (statd)	ONTAP-NAS (NFSv3)	Notifications; may need inbound ephemeral ports (1024-65535) for callbacks.
UDP 4049	Outbound	NFS quota daemon (rquotad)	ONTAP-NAS (NFSv3)	Quotas; skip for v4.
TCP 3260	Outbound	iSCSI target (discovery/data/CHAP)	ONTAP-SAN (iSCSI), E-Series (iSCSI)	Well-known; CHAP auth over this port; enable mutual CHAP for security.

Port/Protocol	Direction	Purpose	Driver/Protocol	Security Notes
TCP 445	Outbound	SMB/CIFS	ONTAP-NAS (SMB), ANF (SMB)	Well-known; use SMB3 with encryption (Trident annotation netapp.io/smb-encryption=true).
TCP/UDP 88 (optional)	Outbound	Kerberos auth	ONTAP (NFS/SMB/iSCSI with Kerb)	If using Kerberos (not default); to AD servers, not storage.
TCP/UDP 389 (optional)	Outbound	LDAP	ONTAP (NFS/SMB with LDAP)	Similar; for name resolution/auth; restrict to AD.

The liveness/readiness probe port can be changed during installation using the `--probe-port` flag. It is important to make sure this port isn't being used by another process on the worker nodes.

Trident REST API

While tridentctl commands and options are the easiest way to interact with the Trident REST API, you can use the REST endpoint directly if you prefer.

When to use the REST API

REST API is useful for advanced installations that use Trident as a standalone binary in non-Kubernetes deployments.

For better security, the Trident `REST API` is restricted to localhost by default when running inside a pod. To change this behavior, you need to set Trident's `-address` argument in its pod configuration.

Using REST API

For examples of how these APIs are called, pass the debug (`-d`) flag. For more information, refer to Manage Trident using tridentctl.

The API works as follows:

GET

`GET <trident-address>/trident/v1/<object-type>`

Lists all objects of that type.

`GET <trident-address>/trident/v1/<object-type>/<object-name>`

Gets the details of the named object.

POST

POST <trident-address>/trident/v1/<object-type>

Creates an object of the specified type.

- Requires a JSON configuration for the object to be created. For the specification of each object type, refer to Manage Trident using tridentctl.
- If the object already exists, behavior varies: backends update the existing object, while all other object types will fail the operation.

DELETE

DELETE <trident-address>/trident/v1/<object-type>/<object-name>

Deletes the named resource.

Volumes associated with backends or storage classes will continue to exist; these must be deleted separately. For more information, refer to Manage Trident using tridentctl.

Command-line options

Trident exposes several command-line options for the Trident orchestrator. You can use these options to modify your deployment.

Logging

-debug

Enables debugging output.

-loglevel <level>

Sets the logging level (debug, info, warn, error, fatal). Defaults to info.

Kubernetes

-k8s_pod

Use this option or `-k8s_api_server` to enable Kubernetes support. Setting this causes Trident to use its containing pod's Kubernetes service account credentials to contact the API server. This only works when Trident runs as a pod in a Kubernetes cluster with service accounts enabled.

-k8s_api_server <insecure-address:insecure-port>

Use this option or `-k8s_pod` to enable Kubernetes support. When specified, Trident connects to the Kubernetes API server using the provided insecure address and port. This Enables Trident to be deployed outside of a pod; however, it only supports insecure connections to the API server. To connect securely, deploy Trident in a pod with the `-k8s_pod` option.

Docker

-volume_driver <name>

Driver name used when registering the Docker plugin. Defaults to `netapp`.

-driver_port <port-number>

Listen on this port rather than a UNIX domain socket.

-config <file>

Required; you must specify this path to a backend configuration file.

REST

-address <ip-or-host>

Specifies the address on which Trident's REST server should listen. Defaults to localhost. When listening on localhost and running inside a Kubernetes pod, the REST interface isn't directly accessible from outside the pod. Use `-address ""` to make the REST interface accessible from the pod IP address.

Trident REST interface can be configured to listen and serve at 127.0.0.1 (for IPv4) or [::1] (for IPv6) only.

-port <port-number>

Specifies the port on which Trident's REST server should listen. Defaults to 8000.

-rest

Enables the REST interface. Defaults to true.

Kubernetes and Trident objects

You can interact with Kubernetes and Trident using REST APIs by reading and writing resource objects. There are several resource objects that dictate the relationship between Kubernetes and Trident, Trident and storage, and Kubernetes and storage. Some of these objects are managed through Kubernetes and the others are managed through Trident.

How do the objects interact with one another?

Perhaps the easiest way to understand the objects, what they are for, and how they interact, is to follow a single request for storage from a Kubernetes user:

1. A user creates a `PersistentVolumeClaim` requesting a new `PersistentVolume` of a particular size from a Kubernetes `StorageClass` that was previously configured by the administrator.
2. The Kubernetes `StorageClass` identifies Trident as its provisioner and includes parameters that tell Trident how to provision a volume for the requested class.
3. Trident looks at its own `StorageClass` with the same name that identifies the matching `Backends` and `StoragePools` that it can use to provision volumes for the class.
4. Trident provisions storage on a matching backend and creates two objects: a `PersistentVolume` in Kubernetes that tells Kubernetes how to find, mount, and treat the volume, and a volume in Trident that retains the relationship between the `PersistentVolume` and the actual storage.
5. Kubernetes binds the `PersistentVolumeClaim` to the new `PersistentVolume`. Pods that include the `PersistentVolumeClaim` mount that PersistentVolume on any host that it runs on.

6. A user creates a `VolumeSnapshot` of an existing PVC, using a `VolumeSnapshotClass` that points to Trident.
7. Trident identifies the volume that is associated with the PVC and creates a snapshot of the volume on its backend. It also creates a `VolumeSnapshotContent` that instructs Kubernetes on how to identify the snapshot.
8. A user can create a `PersistentVolumeClaim` using `VolumeSnapshot` as the source.
9. Trident identifies the required snapshot and performs the same set of steps involved in creating a `PersistentVolume` and a `Volume`.

For further reading about Kubernetes objects, we highly recommend that you read the Persistent Volumes section of the Kubernetes documentation.

Kubernetes `PersistentVolumeClaim` objects

A Kubernetes `PersistentVolumeClaim` object is a request for storage made by a Kubernetes cluster user.

In addition to the standard specification, Trident allows users to specify the following volume-specific annotations if they want to override the defaults that you set in the backend configuration:

Annotation	Volume Option	Supported Drivers
trident.netapp.io/fileSystem	fileSystem	ontap-san, solidfire-san,ontap-san-economy
trident.netapp.io/cloneFromPVC	cloneSourceVolume	ontap-nas, ontap-san, solidfire-san, azure-netapp-files, ontap-san-economy
trident.netapp.io/splitOnClone	splitOnClone	ontap-nas, ontap-san
trident.netapp.io/protocol	protocol	any
trident.netapp.io/exportPolicy	exportPolicy	ontap-nas, ontap-nas-economy, ontap-nas-flexgroup
trident.netapp.io/snapshotPolicy	snapshotPolicy	ontap-nas, ontap-nas-economy, ontap-nas-flexgroup, ontap-san
trident.netapp.io/snapshotReserve	snapshotReserve	ontap-nas, ontap-nas-flexgroup, ontap-san
trident.netapp.io/snapshotDirectory	snapshotDirectory	ontap-nas, ontap-nas-economy, ontap-nas-flexgroup
trident.netapp.io/unixPermissions	unixPermissions	ontap-nas, ontap-nas-economy, ontap-nas-flexgroup
trident.netapp.io/blockSize	blockSize	solidfire-san

Annotation	Volume Option	Supported Drivers
trident.netapp.io/skipRecoveryQueue	skipRecoveryQueue	ontap-nas, ontap-nas-economy, ontap-nas-flexgroup, ontap-san, ontap-san-economy

If the created PV has the `Delete` reclaim policy, Trident deletes both the PV and the backing volume when the PV becomes released (that is, when the user deletes the PVC). Should the delete action fail, Trident marks the PV as such and periodically retries the operation until it succeeds or the PV is manually deleted. If the PV uses the `Retain` policy, Trident ignores it and assumes the administrator will clean it up from Kubernetes and the backend, allowing the volume to be backed up or inspected before its removal. Note that deleting the PV does not cause Trident to delete the backing volume. You should remove it using the REST API (`tridentctl`).

Trident supports the creation of Volume Snapshots using the CSI specification: you can create a Volume Snapshot and use it as a Data Source to clone existing PVCs. This way, point-in-time copies of PVs can be exposed to Kubernetes in the form of snapshots. The snapshots can then be used to create new PVs. Take a look at `On-Demand Volume Snapshots` to see how this would work.

Trident also provides the `cloneFromPVC` and `splitOnClone` annotations for creating clones. You can use these annotations to clone a PVC without having to use the CSI implementation.

Here is an example: If a user already has a PVC called `mysql`, the user can create a new PVC called `mysqlclone` by using the annotation, such as `trident.netapp.io/cloneFromPVC: mysql`. With this annotation set, Trident clones the volume corresponding to the mysql PVC, instead of provisioning a volume from scratch.

Consider the following points:

- NetApp recommends cloning an idle volume.
- A PVC and its clone should be in the same Kubernetes namespace and have the same storage class.
- With the `ontap-nas` and `ontap-san` drivers, it might be desirable to set the PVC annotation `trident.netapp.io/splitOnClone` in conjunction with `trident.netapp.io/cloneFromPVC`. With `trident.netapp.io/splitOnClone` set to `true`, Trident splits the cloned volume from the parent volume and thus, completely decoupling the life cycle of the cloned volume from its parent at the expense of losing some storage efficiency. Not setting `trident.netapp.io/splitOnClone` or setting it to `false` results in reduced space consumption on the backend at the expense of creating dependencies between the parent and clone volumes such that the parent volume cannot be deleted unless the clone is deleted first. A scenario where splitting the clone makes sense is cloning an empty database volume where it's expected for the volume and its clone to greatly diverge and not benefit from storage efficiencies offered by ONTAP.

The `sample-input` directory contains examples of PVC definitions for use with Trident. Refer to Trident `Volume` objects for a full description of the parameters and settings associated with Trident volumes.

Kubernetes `PersistentVolume` objects

A Kubernetes `PersistentVolume` object represents a piece of storage that is made available to the Kubernetes cluster. It has a lifecycle that is independent of the pod that uses it.

Trident creates `PersistentVolume` objects and registers them with the Kubernetes cluster automatically based on the volumes that it provisions. You are not expected to manage them yourself.

When you create a PVC that refers to a Trident-based `StorageClass`, Trident provisions a new volume using the corresponding storage class and registers a new PV for that volume. In configuring the provisioned volume and corresponding PV, Trident follows the following rules:

- Trident generates a PV name for Kubernetes and an internal name that it uses to provision the storage. In both cases, it is assuring that the names are unique in their scope.
- The size of the volume matches the requested size in the PVC as closely as possible, though it might be rounded up to the nearest allocatable quantity, depending on the platform.

Kubernetes `StorageClass` objects

Kubernetes `StorageClass` objects are specified by name in `PersistentVolumeClaims` to provision storage with a set of properties. The storage class itself identifies the provisioner to be used and defines that set of properties in terms the provisioner understands.

It is one of two basic objects that need to be created and managed by the administrator. The other is the Trident backend object.

A Kubernetes `StorageClass` object that uses Trident looks like this:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: <Name>
provisioner: csi.trident.netapp.io
mountOptions: <Mount Options>
parameters: <Trident Parameters>
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

These parameters are Trident-specific and tell Trident how to provision volumes for the class.

The storage class parameters are:

Attribute	Type	Required	Description
attributes	map[string]string	no	See the attributes section below
storagePools	map[string]StringList	no	Map of backend names to lists of storage pools within
additionalStoragePools	map[string]StringList	no	Map of backend names to lists of storage pools within

Attribute	Type	Required	Description
excludeStoragePools	map[string]StringList	no	Map of backend names to lists of storage pools within

Storage attributes and their possible values can be classified into storage pool selection attributes and Kubernetes attributes.

Storage pool selection attributes

These parameters determine which Trident-managed storage pools should be utilized to provision volumes of a given type.

Attribute	Type	Values	Offer	Request	Supported by
media[1]	string	hdd, hybrid, ssd	Pool contains media of this type; hybrid means both	Media type specified	ontap-nas, ontap-nas-economy, ontap-nas-flexgroup, ontap-san, solidfire-san
provisioningType	string	thin, thick	Pool supports this provisioning method	Provisioning method specified	thick: all ontap; thin: all ontap & solidfire-san
backendType	string	ontap-nas, ontap-nas-economy, ontap-nas-flexgroup, ontap-san, solidfire-san, azure-netapp-files, ontap-san-economy	Pool belongs to this type of backend	Backend specified	All drivers
snapshots	bool	true, false	Pool supports volumes with snapshots	Volume with snapshots enabled	ontap-nas, ontap-san, solidfire-san
clones	bool	true, false	Pool supports cloning volumes	Volume with clones enabled	ontap-nas, ontap-san, solidfire-san
encryption	bool	true, false	Pool supports encrypted volumes	Volume with encryption enabled	ontap-nas, ontap-nas-economy, ontap-nas-flexgroups, ontap-san
IOPS	int	positive integer	Pool is capable of guaranteeing IOPS in this range	Volume guaranteed these IOPS	solidfire-san

[1]: Not supported by ONTAP Select systems

In most cases, the values requested directly influence provisioning; for instance, requesting thick provisioning results in a thickly provisioned volume. However, an Element storage pool uses its offered IOPS minimum and maximum to set QoS values, rather than the requested value. In this case, the requested value is used only to select the storage pool.

Ideally, you can use `attributes` alone to model the qualities of the storage you need to satisfy the needs of a particular class. Trident automatically discovers and selects storage pools that match *all* of the `attributes` that you specify.

If you find yourself unable to use `attributes` to automatically select the right pools for a class, you can use the `storagePools` and `additionalStoragePools` parameters to further refine the pools or even to select a specific set of pools.

You can use the `storagePools` parameter to further restrict the set of pools that match any specified `attributes`. In other words, Trident uses the intersection of pools identified by the `attributes` and `storagePools` parameters for provisioning. You can use either parameter alone or both together.

You can use the `additionalStoragePools` parameter to extend the set of pools that Trident uses for provisioning, regardless of any pools selected by the `attributes` and `storagePools` parameters.

You can use the `excludeStoragePools` parameter to filter the set of pools that Trident uses for provisioning. Using this parameter removes any pools that match.

In the `storagePools` and `additionalStoragePools` parameters, each entry takes the form `<backend>:<storagePoolList>`, where `<storagePoolList>` is a comma-separated list of storage pools for the specified backend. For example, a value for `additionalStoragePools` might look like `ontapnas_192.168.1.100:aggr1,aggr2;solidfire_192.168.1.101:bronze`.
These lists accept regex values for both the backend and list values. You can use `tridentctl get backend` to get the list of backends and their pools.

Kubernetes attributes

These attributes have no impact on the selection of storage pools/backends by Trident during dynamic provisioning. Instead, these attributes simply supply parameters supported by Kubernetes Persistent Volumes. Worker nodes are responsible for filesystem create operations and might require filesystem utilities, such as xfsprogs.

Attribute	Type	Values	Description	Relevant Drivers	Kubernetes Version
fsType	string	ext4, ext3, xfs	The file system type for block volumes	solidfire-san, ontap-nas, ontap-nas-economy, ontap-nas-flexgroup, ontap-san, ontap-san-economy	All

Attribute	Type	Values	Description	Relevant Drivers	Kubernetes Version
allowVolumeExpansion	boolean	true, false	Enable or disable support for growing the PVC size	ontap-nas, ontap-nas-economy, ontap-nas-flexgroup, ontap-san, ontap-san-economy, solidfire-san, azure-netapp-files	1.11+
volumeBindingMode	string	Immediate, WaitForFirstConsumer	Choose when volume binding and dynamic provisioning occurs	All	1.19 - 1.26

- The `fsType` parameter is used to control the desired file system type for SAN LUNs. In addition, Kubernetes also uses the presence of `fsType` in a storage class to indicate a filesystem exists. Volume ownership can be controlled using the `fsGroup` security context of a pod only if `fsType` is set. Refer to Kubernetes: Configure a Security Context for a Pod or Container for an overview on setting volume ownership using the `fsGroup` context. Kubernetes will apply the `fsGroup` value only if:
  - `fsType` is set in the storage class.
  - The PVC access mode is RWO.

  For NFS storage drivers, a filesystem already exists as part of the NFS export. In order to use `fsGroup` the storage class still needs to specify a `fsType`. You can set it to `nfs` or any non-null value.
- Refer to Expand volumes for further details on volume expansion.
- The Trident installer bundle provides several example storage class definitions for use with Trident in `sample-input/storage-class-*.yaml`. Deleting a Kubernetes storage class causes the corresponding Trident storage class to be deleted as well.

Kubernetes `VolumeSnapshotClass` objects

Kubernetes `VolumeSnapshotClass` objects are analogous to `StorageClasses`. They help define multiple classes of storage and are referenced by volume snapshots to associate the snapshot with the required snapshot class. Each volume snapshot is associated with a single volume snapshot class.

A `VolumeSnapshotClass` should be defined by an administrator in order to create snapshots. A volume snapshot class is created with the following definition:

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

The `driver` specifies to Kubernetes that requests for volume snapshots of the `csi-snapclass` class are handled by Trident. The `deletionPolicy` specifies the action to be taken when a snapshot must be deleted. When `deletionPolicy` is set to `Delete`, the volume snapshot objects as well as the underlying snapshot on the storage cluster are removed when a snapshot is deleted. Alternatively, setting it to `Retain` means that `VolumeSnapshotContent` and the physical snapshot are retained.

Kubernetes `VolumeSnapshot` objects

A Kubernetes `VolumeSnapshot` object is a request to create a snapshot of a volume. Just as a PVC represents a request made by a user for a volume, a volume snapshot is a request made by a user to create a snapshot of an existing PVC.

When a volume snapshot request comes in, Trident automatically manages the creation of the snapshot for the volume on the backend and exposes the snapshot by creating a unique `VolumeSnapshotContent` object. You can create snapshots from existing PVCs and use the snapshots as a DataSource when creating new PVCs.

The lifecycle of a VolumeSnapshot is independent of the source PVC: a snapshot persists even after the source PVC is deleted. When deleting a PVC which has associated snapshots, Trident marks the backing volume for this PVC in a **Deleting** state, but does not remove it completely. The volume is removed when all the associated snapshots are deleted.

Kubernetes `VolumeSnapshotContent` objects

A Kubernetes `VolumeSnapshotContent` object represents a snapshot taken from an already provisioned volume. It is analogous to a `PersistentVolume` and signifies a provisioned snapshot on the storage cluster. Similar to `PersistentVolumeClaim` and `PersistentVolume` objects, when a snapshot is created, the `VolumeSnapshotContent` object maintains a one-to-one mapping to the `VolumeSnapshot` object, which had requested the snapshot creation.

The `VolumeSnapshotContent` object contains details that uniquely identify the snapshot, such as the `snapshotHandle`. This `snapshotHandle` is a unique combination of the name of the PV and the name of the `VolumeSnapshotContent` object.

When a snapshot request comes in, Trident creates the snapshot on the backend. After the snapshot is created, Trident configures a `VolumeSnapshotContent` object and thus exposes the snapshot to the Kubernetes API.

Typically, you do not need to manage the `VolumeSnapshotContent` object. An exception to this is when you want to import a volume snapshot created outside of Trident.

Kubernetes `VolumeGroupSnapshotClass` objects

Kubernetes `VolumeGroupSnapshotClass` objects are analogous to `VolumeSnapshotClass`. They help define multiple classes of storage and are referenced by volume group snapshots to associate the snapshot with the required snapshot class. Each volume group snapshot is associated with a single volume group snapshot class.

A `VolumeGroupSnapshotClass` should be defined by an administrator in order to create group of snapshots. A volume group snapshot class is created with the following definition:

```
apiVersion: groupsnapshot.storage.k8s.io/v1beta1
kind: VolumeGroupSnapshotClass
metadata:
  name: csi-group-snap-class
  annotations:
    kubernetes.io/description: "Trident group snapshot class"
driver: csi.trident.netapp.io
deletionPolicy: Delete
```

The `driver` specifies to Kubernetes that requests for volume group snapshots of the `csi-group-snap-class` class are handled by Trident. The `deletionPolicy` specifies the action to be taken when a group snapshot must be deleted. When `deletionPolicy` is set to `Delete`, the volume group snapshot objects as well as the underlying snapshot on the storage cluster are removed when a snapshot is deleted. Alternatively, setting it to `Retain` means that `VolumeGroupSnapshotContent` and the physical snapshot are retained.

Kubernetes `VolumeGroupSnapshot` objects

A Kubernetes `VolumeGroupSnapshot` object is a request to create a snapshot of a multiple volumes. Just as a PVC represents a request made by a user for a volume, a volume group snapshot is a request made by a user to create a snapshot of an existing PVC.

When a volume group snapshot request comes in, Trident automatically manages the creation of the group snapshot for the volumes on the backend and exposes the snapshot by creating a unique `VolumeGroupSnapshotContent` object. You can create snapshots from existing PVCs and use the snapshots as a DataSource when creating new PVCs.

> The lifecycle of a VolumeGroupSnapshot is independent of the source PVC: a snapshot persists even after the source PVC is deleted. When deleting a PVC which has associated snapshots, Trident marks the backing volume for this PVC in a **Deleting** state, but does not remove it completely. The volume group snapshot is removed when all the associated snapshots are deleted.

Kubernetes `VolumeGroupSnapshotContent` objects

A Kubernetes `VolumeGroupSnapshotContent` object represents a group snapshot taken from an already provisioned volume. It is analogous to a `PersistentVolume` and signifies a provisioned snapshot on the storage cluster. Similar to `PersistentVolumeClaim` and `PersistentVolume` objects, when a snapshot is created, the `VolumeSnapshotContent` object maintains a one-to-one mapping to the `VolumeSnapshot` object, which had requested the snapshot creation.

The `VolumeGroupSnapshotContent` object contains details that identify the snapshot group, such as the `volumeGroupSnapshotHandle` and and individual volumeSnapshotHandles existing on the storage system.

When a snapshot request comes in, Trident creates the volume group snapshot on the backend. After the volume group snapshot is created, Trident configures a `VolumeGroupSnapshotContent` object and thus exposes the snapshot to the Kubernetes API.

Kubernetes `CustomResourceDefinition` objects

Kubernetes Custom Resources are endpoints in the Kubernetes API that are defined by the administrator and are used to group similar objects. Kubernetes supports the creation of custom resources for storing a collection of objects. You can obtain these resource definitions by running `kubectl get crds`.

Custom Resource Definitions (CRDs) and their associated object metadata are stored by Kubernetes in its metadata store. This eliminates the need for a separate store for Trident.

Trident uses `CustomResourceDefinition` objects to preserve the identity of Trident objects, such as Trident backends, Trident storage classes, and Trident volumes. These objects are managed by Trident. In addition, the CSI volume snapshot framework introduces some CRDs that are required to define volume snapshots.

CRDs are a Kubernetes construct. Objects of the resources defined above are created by Trident. As a simple example, when a backend is created using `tridentctl`, a corresponding `tridentbackends` CRD object is created for consumption by Kubernetes.

Here are a few points to keep in mind about Trident's CRDs:

- When Trident is installed, a set of CRDs are created and can be used like any other resource type.
- When uninstalling Trident by using the `tridentctl uninstall` command, Trident pods are deleted but the created CRDs are not cleaned up. Refer to Uninstall Trident to understand how Trident can be completely removed and reconfigured from scratch.

Trident `StorageClass` objects

Trident creates matching storage classes for Kubernetes `StorageClass` objects that specify `csi.trident.netapp.io` in their provisioner field. The storage class name matches that of the Kubernetes `StorageClass` object it represents.

With Kubernetes, these objects are created automatically when a Kubernetes `StorageClass` that uses Trident as a provisioner is registered.

Storage classes comprise a set of requirements for volumes. Trident matches these requirements with the attributes present in each storage pool; if they match, that storage pool is a valid target for provisioning volumes using that storage class.

You can create storage class configurations to directly define storage classes by using the REST API. However, for Kubernetes deployments, we expect them to be created when registering new Kubernetes `StorageClass` objects.

Trident backend objects

Backends represent the storage providers on top of which Trident provisions volumes; a single Trident instance can manage any number of backends.

This is one of the two object types that you create and manage yourself. The other is the Kubernetes `StorageClass` object.

For more information about how to construct these objects, refer to configuring backends.

Trident `StoragePool` objects

Storage pools represent the distinct locations available for provisioning on each backend. For ONTAP, these correspond to aggregates in SVMs. For NetApp HCI/SolidFire, these correspond to administrator-specified QoS bands. Each storage pool has a set of distinct storage attributes, which define its performance characteristics and data protection characteristics.

Unlike the other objects here, storage pool candidates are always discovered and managed automatically.

Trident `Volume` objects

Volumes are the basic unit of provisioning, comprising backend endpoints, such as NFS shares, and iSCSI and FC LUNs. In Kubernetes, these correspond directly to `PersistentVolumes`. When you create a volume, ensure that it has a storage class, which determines where that volume can be provisioned, along with a size.

- In Kubernetes, these objects are managed automatically. You can view them to see what Trident provisioned.
- When deleting a PV with associated snapshots, the corresponding Trident volume is updated to a **Deleting** state. For the Trident volume to be deleted, you should remove the snapshots of the volume.

A volume configuration defines the properties that a provisioned volume should have.

Attribute	Type	Required	Description
version	string	no	Version of the Trident API ("1")
name	string	yes	Name of volume to create
storageClass	string	yes	Storage class to use when provisioning the volume
size	string	yes	Size of the volume to provision in bytes
protocol	string	no	Protocol type to use; "file" or "block"
internalName	string	no	Name of the object on the storage system; generated by Trident

Attribute	Type	Required	Description
cloneSourceVolume	string	no	ontap (nas, san) & solidfire-*: Name of the volume to clone from
splitOnClone	string	no	ontap (nas, san): Split the clone from its parent
snapshotPolicy	string	no	ontap-*: Snapshot policy to use
snapshotReserve	string	no	ontap-*: Percentage of volume reserved for snapshots
exportPolicy	string	no	ontap-nas*: Export policy to use
snapshotDirectory	bool	no	ontap-nas*: Whether the snapshot directory is visible
unixPermissions	string	no	ontap-nas*: Initial UNIX permissions
blockSize	string	no	solidfire-*: Block/sector size
fileSystem	string	no	File system type
skipRecoveryQueue	string	no	During volume deletion, bypass the recovery queue in storage and delete the volume immediately.

Trident generates `internalName` when creating the volume. This consists of two steps. First, it prepends the storage prefix (either the default `trident` or the prefix in the backend configuration) to the volume name, resulting in a name of the form `<prefix>-<volume-name>`. It then proceeds to sanitize the name, replacing characters not permitted in the backend. For ONTAP backends, it replaces hyphens with underscores (thus, the internal name becomes `<prefix>_<volume-name>`). For Element backends, it replaces underscores with hyphens.

You can use volume configurations to directly provision volumes using the REST API, but in Kubernetes deployments we expect most users to use the standard Kubernetes `PersistentVolumeClaim` method. Trident creates this volume object automatically as part of the provisioning process.

Trident `Snapshot` objects

Snapshots are a point-in-time copy of volumes, which can be used to provision new volumes or restore state. In Kubernetes, these correspond directly to `VolumeSnapshotContent` objects. Each snapshot is associated with a volume, which is the source of the data for the snapshot.

Each `Snapshot` object includes the properties listed below:

Attribute	Type	Required	Description
version	String	Yes	Version of the Trident API ("1")
name	String	Yes	Name of the Trident snapshot object
internalName	String	Yes	Name of the Trident snapshot object on the storage system
volumeName	String	Yes	Name of the Persistent Volume for which the snapshot is created
volumeInternalName	String	Yes	Name of the associated Trident volume object on the storage system

In Kubernetes, these objects are managed automatically. You can view them to see what Trident provisioned.

When a Kubernetes `VolumeSnapshot` object request is created, Trident works by creating a snapshot object on the backing storage system. The `internalName` of this snapshot object is generated by combining the prefix `snapshot-` with the `UID` of the `VolumeSnapshot` object (for example, `snapshot-e8d8a0ca-9826-11e9-9807-525400f3f660`). `volumeName` and `volumeInternalName` are populated by getting the details of the backing
volume.

Trident `ResourceQuota` object

The Trident deamonset consumes a `system-node-critical` Priority Class—the highest Priority Class available in Kubernetes—to ensure Trident can identify and clean up volumes during graceful node shutdown and allow Trident daemonset pods to preempt workloads with a lower priority in clusters where there is high resource pressure.

To accomplish this, Trident employs a `ResourceQuota` object to ensure a "system-node-critical" Priority Class on the Trident daemonset is satisfied. Prior to deployment and daemonset creation, Trident looks for the `ResourceQuota` object and, if not discovered, applies it.

If you need more control over the default Resource Quota and Priority Class, you can generate a `custom.yaml` or configure the `ResourceQuota` object using Helm chart.

The following is an example of a `ResourceQuota`object prioritizing the Trident daemonset.

```
apiVersion: <version>
kind: ResourceQuota
metadata:
  name: trident-csi
  labels:
    app: node.csi.trident.netapp.io
spec:
  scopeSelector:
    matchExpressions:
      - operator: In
        scopeName: PriorityClass
        values:
          - system-node-critical
```

For more information on Resource Quotas, refer to Kubernetes: Resource Quotas.

Clean up `ResourceQuota` if installation fails

In the rare case where installation fails after the `ResourceQuota` object is created, first try uninstalling and then reinstall.

If that doesn't work, manually remove the `ResourceQuota` object.

Remove `ResourceQuota`

If you prefer to control your own resource allocation, you can remove the Trident `ResourceQuota` object using the command:

```
kubectl delete quota trident-csi -n trident
```

Pod Security Standards (PSS) and Security Context Constraints (SCC)

Kubernetes Pod Security Standards (PSS) and Pod Security Policies (PSP) define permission levels and restrict the behavior of pods. OpenShift Security Context Constraints (SCC) similarly define pod restriction specific to the OpenShift Kubernetes Engine. To provide this customization, Trident enables certain permissions during installation. The following sections detail the permissions set by Trident.

PSS replaces Pod Security Policies (PSP). PSP was deprecated in Kubernetes v1.21 and will be removed in v1.25. For more information, Refer to Kubernetes: Security.

Required Kubernetes Security Context and Related Fields

Permission	Description
Privileged	CSI requires mount points to be Bidirectional, which means the Trident node pod must run a privileged container. For more information, refer to Kubernetes: Mount propagation.
Host networking	Required for the iSCSI daemon. `iscsiadm` manages iSCSI mounts and uses host networking to communicate with the iSCSI daemon.
Host IPC	NFS uses interprocess communication (IPC) to communicate with the NFSD.
Host PID	Required to start `rpc-statd` for NFS. Trident queries host processes to determine if `rpc-statd` is running before mounting NFS volumes.

Capabilities	The `SYS_ADMIN` capability is provided as part of the default capabilities for privileged containers. For example, Docker sets these capabilities for privileged containers:
	`CapPrm: 0000003fffffffff`
	`CapEff: 0000003fffffffff`

Seccomp	Seccomp profile is always "Unconfined" in privileged containers; therefore, it cannot be enabled in Trident.
SELinux	On OpenShift, privileged containers are run in the `spc_t` ("Super Privileged Container") domain, and unprivileged containers are run in the `container_t` domain. On `containerd`, with `container-selinux` installed, all containers are run in the `spc_t` domain, which effectively disables SELinux. Therefore, Trident does not add `seLinuxOptions` to containers.
DAC	Privileged containers must be run as root. Non-privileged containers run as root to access unix sockets required by CSI.

Pod Security Standards (PSS)

Label	Description	Default

`pod-security.kubernetes.io/enforce`
 	
`pod-security.kubernetes.io/enforce-version`
	Allows the Trident Controller and nodes to be admitted into the install namespace.
	
	Do not change the namespace label.
		`enforce: privileged`
		
		`enforce-version: <version of the current cluster or highest version of PSS tested.>`

Changing the namespace labels can result in pods not being scheduled, an "Error creating: …" or, "Warning: trident-csi-…". If this happens, check if the namespace label for `privileged` was changed. If so, reinstall Trident.

Pod Security Policies (PSP)

Field	Description	Default
`allowPrivilegeEscalation`	Privileged containers must allow privilege escalation.	`true`
`allowedCSIDrivers`	Trident does not use inline CSI ephemeral volumes.	Empty
`allowedCapabilities`	Non-privileged Trident containers do not require more capabilities than the default set and privileged containers are granted all possible capabilities.	Empty
`allowedFlexVolumes`	Trident does not make use of a FlexVolume driver, therefore they are not included in the list of allowed volumes.	Empty
`allowedHostPaths`	The Trident node pod mounts the node's root filesystem, therefore there is no benefit to setting this list.	Empty
`allowedProcMountTypes`	Trident does not use any `ProcMountTypes`.	Empty
`allowedUnsafeSysctls`	Trident does not require any unsafe `sysctls`.	Empty
`defaultAddCapabilities`	No capabilities are required to be added to privileged containers.	Empty
`defaultAllowPrivilegeEscalation`	Allowing privilege escalation is handled in each Trident pod.	`false`
`forbiddenSysctls`	No `sysctls` are allowed.	Empty
`fsGroup`	Trident containers run as root.	`RunAsAny`
`hostIPC`	Mounting NFS volumes requires host IPC to communicate with `nfsd`	`true`
`hostNetwork`	iscsiadm requires the host network to communicate with the iSCSI daemon.	`true`
`hostPID`	Host PID is required to check if `rpc-statd` is running on the node.	`true`
`hostPorts`	Trident does not use any host ports.	Empty

Field	Description	Default
`privileged`	Trident node pods must run a privileged container in order to mount volumes.	`true`
`readOnlyRootFilesystem`	Trident node pods must write to the node filesystem.	`false`
`requiredDropCapabilities`	Trident node pods run a privileged container and cannot drop capabilities.	`none`
`runAsGroup`	Trident containers run as root.	`RunAsAny`
`runAsUser`	Trident containers run as root.	`runAsAny`
`runtimeClass`	Trident does not use `RuntimeClasses`.	Empty
`seLinux`	Trident does not set `seLinuxOptions` because there are currently differences in how container runtimes and Kubernetes distributions handle SELinux.	Empty
`supplementalGroups`	Trident containers run as root.	`RunAsAny`
`volumes`	Trident pods require these volume plugins.	`hostPath, projected, emptyDir`

Security Context Constraints (SCC)

Labels	Description	Default
`allowHostDirVolumePlugin`	Trident node pods mount the node's root filesystem.	`true`
`allowHostIPC`	Mounting NFS volumes requires host IPC to communicate with `nfsd`.	`true`
`allowHostNetwork`	iscsiadm requires the host network to communicate with the iSCSI daemon.	`true`
`allowHostPID`	Host PID is required to check if `rpc-statd` is running on the node.	`true`
`allowHostPorts`	Trident does not use any host ports.	`false`
`allowPrivilegeEscalation`	Privileged containers must allow privilege escalation.	`true`
`allowPrivilegedContainer`	Trident node pods must run a privileged container in order to mount volumes.	`true`

Labels	Description	Default
`allowedUnsafeSysctls`	Trident does not require any unsafe `sysctls`.	`none`
`allowedCapabilities`	Non-privileged Trident containers do not require more capabilities than the default set and privileged containers are granted all possible capabilities.	Empty
`defaultAddCapabilities`	No capabilities are required to be added to privileged containers.	Empty
`fsGroup`	Trident containers run as root.	`RunAsAny`
`groups`	This SCC is specific to Trident and is bound to its user.	Empty
`readOnlyRootFilesystem`	Trident node pods must write to the node filesystem.	`false`
`requiredDropCapabilities`	Trident node pods run a privileged container and cannot drop capabilities.	`none`
`runAsUser`	Trident containers run as root.	`RunAsAny`
`seLinuxContext`	Trident does not set `seLinuxOptions` because there are currently differences in how container runtimes and Kubernetes distributions handle SELinux.	Empty
`seccompProfiles`	Privileged containers always run "Unconfined".	Empty
`supplementalGroups`	Trident containers run as root.	`RunAsAny`
`users`	One entry is provided to bind this SCC to the Trident user in the Trident namespace.	n/a
`volumes`	Trident pods require these volume plugins.	`hostPath, downwardAPI, projected, emptyDir`

Copyright information

Copyright © 2026 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.