Trident Documentation

NetApp

Jan 31, 2019

Introduction

Storage Orchestrator for Containers

CHAPTER 1

What is Trident?

Trident is a fully supported open source project maintained by NetApp. It has been designed from the ground up to help you meet the sophisticated persistence demands of your containerized applications.

Through its support for popular container platforms like Kubernetes and Docker, Trident understands the natural and evolving languages of those platforms, and translates requirements expressed or implied through them into an automated and orchestrated response from the infrastructure.

Today, that infrastructure includes our ONTAP (AFF/FAS/Select/Cloud), Element (HCI/SolidFire), and SANtricity (E/EF-Series) data management software. That list continues to grow.

CHAPTER 2

Trident for Kubernetes

Trident integrates natively with Kubernetes and its Persistent Volume framework to seamlessly provision and manage volumes from systems running any combination of NetApp's ONTAP (AFF/FAS/Select/Cloud), Element (HCI/SolidFire), or SANtricity (E/EF-Series) data management platforms.

Relative to other Kubernetes provisioners, Trident is novel in the following respects:

1. It is the first out-of-tree, out-of-process storage provisioner that works by watching events at the Kubernetes API Server, affording it levels of visibility and flexibility that cannot otherwise be achieved.

2. It is capable of orchestrating across multiple platforms at the same time through a unified interface. Rather than tying a request for a persistent volume to a particular system, Trident selects one from those it manages based on the higher-level qualities that the user is looking for in their volume.

Trident tightly integrates with Kubernetes to allow your users to request and manage persistent volumes using native Kubernetes interfaces and constructs. It's designed to work in such a way that your users can take advantage of the underlying capabilities of your storage infrastructure without having to know anything about it.

It automates the rest for you, the Kubernetes administrator, based on policies that you define.

A great way to get a feel for what we're trying to accomplish is to see Trident in action from the perspective of an end user. This is a great demonstration of Kubernetes volume consumption when Trident is in the mix, through the lens of Red Hat's OpenShift platform, which is itself built on Kubernetes. All of the concepts in the video apply to any Kubernetes deployment.

While some details about Trident and NetApp storage systems are shown in the video to help you see what's going on behind-the-scenes, in standard deployments Trident and the rest of the infrastructure is completely hidden from the user.

Let's lift up the covers a bit to better understand Trident and what it is doing. This introductory video provides a great way to do just that.

2.1 Deploying

This guide will take you through the process of deploying Trident and provisioning your first volume automatically.

2.1.1 Before you begin

If you have not already familiarized yourself with the *basic concepts*, now is a great time to do that. Go ahead, we'll be here when you get back.

To deploy Trident you need:

Need Kubernetes?

If you do not already have a Kubernetes cluster, you can easily create one for demonstration purposes using our *simple Kubernetes install guide*.

- Full privileges to a *supported Kubernetes cluster*
- Access to a *supported NetApp storage system*
- *Volume mount capability* from all of the Kubernetes worker nodes
- A Linux host with `kubectl` (or `oc`, if you're using OpenShift) installed and configured to manage the Kubernetes cluster you want to use
- If you are using Kubernetes with Docker EE 2.1, follow their steps to enable CLI access.

Got all that? Great! Let's get started.

2.1.2 1: Qualify your Kubernetes cluster

You made sure that you have everything in hand from the *previous section*, right? Right.

The first thing you need to do is log into the Linux host and verify that it is managing a *working*, *supported Kubernetes cluster* that you have the necessary privileges to.

Note: With OpenShift, you will use `oc` instead of `kubectl` in all of the examples that follow, and you need to login as **system:admin** first by running `oc login -u system:admin`.

```
# Are you running a supported Kubernetes server version?
kubectl version

# Are you a Kubernetes cluster administrator?
kubectl auth can-i '*' '*' --all-namespaces

# Can you launch a pod that uses an image from Docker Hub and can reach your
# storage system over the pod network?
kubectl run -i --tty ping --image=busybox --restart=Never --rm -- \
  ping <management IP>
```

2.1.3 2: Download & extract the installer

Download the latest version of the Trident installer bundle from the *Downloads* section and extract it.

For example, if the latest version is 19.01.0:

```
wget https://github.com/NetApp/trident/releases/download/v19.01.0/trident-installer-
↪19.01.0.tar.gz
tar -xf trident-installer-19.01.0.tar.gz
cd trident-installer
```

2.1.4 3: Configure the installer

Why does Trident need an installer?

We have an interesting chicken/egg problem: how to make it easy to get a persistent volume to store Trident's own metadata when Trident itself isn't running yet. The installer handles that for you!

Configure a storage backend that the Trident installer will use to provision a volume to store its own metadata.

You do this by placing a `backend.json` file in the installer's `setup` directory. Sample configuration files for different backend types can be found in the `sample-input` directory.

Visit the *backend configuration* section of this guide for more details about how to craft the configuration file for your backend type.

Note: Many of the backends require some *basic preparation*, so make sure that's been done before you try to use it. Also, we don't recommend an ontap-nas-economy backend or ontap-nas-flexgroup backend for this step as volumes of these types have specialized and limited capabilities relative to the volumes provisioned on other types of backends.

```
cp sample-input/<backend template>.json setup/backend.json
# Fill out the template for your backend
vi setup/backend.json
```

2.1.5 4: Install Trident

First, let's verify that Trident can be installed:

```
./tridentctl install --dry-run -n trident
INFO Starting storage driver.                backend=setup/backend.json
INFO Storage driver loaded.                  driver=ontap-nas
INFO Dry run completed, no problems found.
```

The `--dry-run` argument tells the installer to inspect the current environment and checks that everything looks good for a Trident installation, but it makes no changes to the environment and will *not* install Trident.

The `-n` argument specifies the namespace (project in OpenShift) that Trident will be installed into. We recommend installing Trident into its own namespace to isolate it from other applications.

Provided that everything was configured correctly, you can now run the Trident installer and it should be running in a few minutes:

```
./tridentctl install -n trident
INFO Starting storage driver.                backend=setup/backend.json
INFO Storage driver loaded.                  driver=ontap-nas
INFO Starting Trident installation.          namespace=trident
INFO Created service account.
```

(continues on next page)

(continued from previous page)

```
INFO Created cluster role.
INFO Created cluster role binding.
INFO Created PVC.
INFO Created PV.                          pv=trident
INFO Waiting for PVC to be bound.         pvc=trident
INFO Created Trident deployment.
INFO Waiting for Trident pod to start.
INFO Trident pod started.                 namespace=trident pod=trident-7d5d659bd7-
↪tzth6
INFO Trident installation succeeded.
```

It will look like this when the installer is complete:

```
kubectl get pod -n trident
NAME                        READY     STATUS    RESTARTS   AGE
trident-7d5d659bd7-tzth6    2/2       Running   1          14s

./tridentctl -n trident version
+----------------+----------------+
| SERVER VERSION | CLIENT VERSION |
+----------------+----------------+
| 19.01.0        | 19.01.0        |
+----------------+----------------+
```

If that's what you see, you're done with this step, but **Trident is not yet fully configured.** Go ahead and continue to the next step.

However, if the installer does not complete successfully or you don't see a **Running** `trident-<generated id>`, then Trident had a problem and the platform was *not* installed.

To help figure out what went wrong, you could run the installer again using the `-d` argument, which will turn on debug mode and help you understand what the problem is:

```
./tridentctl install -n trident -d
```

After addressing the problem, you can clean up the installation and go back to the beginning of this step by first running:

```
./tridentctl uninstall -n trident
INFO Deleted Trident deployment.
INFO Deleted cluster role binding.
INFO Deleted cluster role.
INFO Deleted service account.
INFO Removed Trident user from security context constraint.
INFO Trident uninstallation succeeded.
```

If you continue to have trouble, visit the *troubleshooting guide* for more advice.

Customized Installation

Trident's installer allows you to customize attributes such as PV or PVC default names, by using the installer's `--pv` or `--pvc` parameters. You can also specify a storage volume name and size by using `--volume-name` and `--volume-size`. If you have copied the Trident images to a private repository, you can specify the image names by using `--trident-image` and `--etcd-image`.

Users can also customize Trident's deployment files. Using the `--generate-custom-yaml` parameter will create the following YAML files in the installer's `setup` directory:

- trident-clusterrolebinding.yaml
- trident-deployment.yaml
- trident-pvc.yaml
- trident-clusterrole.yaml
- trident-namespace.yaml
- trident-serviceaccount.yaml

Once you have generated these files, you can modify them according to your needs and then use the `--use-custom-yaml` to install a customized version of Trident.

```
./tridentctl install -n trident --use-custom-yaml --volume-name my_volume
```

2.1.6 5: Verify your first backend

You already created a *backend* in step 3 to provision a volume for that Trident uses for its own metadata.

During a first-time installation, the installer assumes you want to use that backend for the rest of the volumes that Trident provisions.

```
./tridentctl -n trident get backend
+----------------------+----------------+--------+---------+
|         NAME         | STORAGE DRIVER | ONLINE | VOLUMES |
+----------------------+----------------+--------+---------+
| ontapnas_10.0.0.1    | ontap-nas      | true   |       0 |
+----------------------+----------------+--------+---------+
```

You can add more backends, or replace the initial one with other backends (it won't affect the volume where Trident keeps its metadata). It's up to you.

```
./tridentctl -n trident create backend -f <path-to-backend-config-file>
+----------------------+----------------+--------+---------+
|         NAME         | STORAGE DRIVER | ONLINE | VOLUMES |
+----------------------+----------------+--------+---------+
| ontapnas_10.0.1.1    | ontap-nas      | true   |       0 |
+----------------------+----------------+--------+---------+
```

If the creation fails, something was wrong with the backend configuration. You can view the logs to determine the cause by running:

```
./tridentctl -n trident logs
```

After addressing the problem, simply go back to the beginning of this step and try again. If you continue to have trouble, visit the *troubleshooting guide* for more advice on how to determine what went wrong.

2.1.7 6: Add your first storage class

Kubernetes users provision volumes using persistent volume claims (PVCs) that specify a storage class by name. The details are hidden from users, but a storage class identifies the provisioner that will be used for that class (in this case, Trident) and what that class means to the provisioner.

Basic too basic?

This is just a basic storage class to get you started. There's an art to *crafting differentiated storage classes* that you should explore further when you're looking at building them for production.

Create a storage class Kubernetes users will specify when they want a volume. The configuration of the class needs to model the backend that you created in the previous step so that Trident will use it to provision new volumes.

The simplest storage class to start with is one based on the `sample-input/storage-class-basic.yaml.templ` file that comes with the installer, replacing `__BACKEND_TYPE__` with the storage driver name.

```
./tridentctl -n trident get backend
+----------------------+----------------+--------+---------+
|         NAME         | STORAGE DRIVER | ONLINE | VOLUMES |
+----------------------+----------------+--------+---------+
| ontapnas_10.0.0.1    | ontap-nas      | true   |       0 |
+----------------------+----------------+--------+---------+

cp sample-input/storage-class-basic.yaml.templ sample-input/storage-class-basic.yaml

# Modify __BACKEND_TYPE__ with the storage driver field above (e.g., ontap-nas)
vi sample-input/storage-class-basic.yaml
```

This is a Kubernetes object, so you will use `kubectl` to create it in Kubernetes.

```
kubectl create -f sample-input/storage-class-basic.yaml
```

You should now see a **basic** storage class in both Kubernetes and Trident, and Trident should have discovered the pools on the backend.

```
kubectl get sc basic
NAME      PROVISIONER
basic     netapp.io/trident

./tridentctl -n trident get storageclass basic -o json
{
  "items": [
    {
      "Config": {
        "version": "1",
        "name": "basic",
        "attributes": {
          "backendType": "ontap-nas"
        }
      },
      "storage": {
        "ontapnas_10.0.0.1": [
          "aggr1",
          "aggr2",
          "aggr3",
          "aggr4"
        ]
      }
    }
  ]
}
```

2.1.8 7: Provision your first volume

Now you're ready to dynamically provision your first volume. How exciting! This is done by creating a Kubernetes persistent volume claim (PVC) object, and this is exactly how your users will do it too.

Create a persistent volume claim (PVC) for a volume that uses the storage class that you just created.

See `sample-input/pvc-basic.yaml` for an example. Make sure the storage class name matches the one that you created in 6.

```
kubectl create -f sample-input/pvc-basic.yaml

# The '-aw' argument lets you watch the pvc get provisioned
kubectl get pvc -aw
NAME      STATUS    VOLUME    CAPACITY   ACCESS MODES   STORAGECLASS   AGE
basic     Pending                                       basic          1s
basic     Pending   default-basic-6cb59   0                   basic     5s
basic     Bound     default-basic-6cb59   1Gi        RWO      basic     5s
```

2.1.9 8: Mount the volume in a pod

Now that you have a volume, let's mount it. We'll launch an nginx pod that mounts the PV under `/usr/share/nginx/html`.

```
cat << EOF > task-pv-pod.yaml
kind: Pod
apiVersion: v1
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
       claimName: basic
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
EOF
kubectl create -f task-pv-pod.yaml
```

```
# Wait for the pod to start
kubectl get pod -aw

# Verify that the volume is mounted on /usr/share/nginx/html
kubectl exec -it task-pv-pod -- df -h /usr/share/nginx/html
Filesystem                                      Size  Used Avail Use% Mounted on
10.0.0.1:/trident_demo_default_basic_6cb59  973M  192K  973M   1% /usr/share/nginx/
↪html

# Delete the pod
kubectl delete pod task-pv-pod
```

At this point the pod (application) no longer exists but the volume is still there. You could use it from another pod if you wanted to.

To delete the volume, simply delete the claim:

```
kubectl delete pvc basic
```

Check you out! You did it! Now you're dynamically provisioning Kubernetes volumes like a boss.

2.2 Common tasks

2.2.1 Managing Trident

Installing Trident

Follow the extensive *deployment* guide.

Updating Trident

The best way to update to the latest version of Trident is to download the latest installer bundle and run:

```
./tridentctl uninstall -n <namespace>
./tridentctl install   -n <namespace>
```

By default the uninstall command will leave all of Trident's state intact by not deleting the PVC and PV used by the Trident deployment, allowing an uninstall followed by an install to act as an upgrade.

PVs that have already been provisioned will remain available while Trident is offline, and Trident will provision volumes for any PVCs that are created in the interim once it is back online.

Note: When upgrading from Trident 18.10 with Docker EE 2.0, you must use the Trident 18.10 installer (specifying the `--ucp-host` and `--ucp-bearer-token` parameters) to uninstall Trident. See the 18.10 docs for details.

Uninstalling Trident

The uninstall command in tridentctl will remove all of the resources associated with Trident except for the PVC, PV and backing volume, making it easy to run the installer again to update to a more recent version.

```
./tridentctl uninstall -n <namespace>
```

To fully uninstall Trident and remove the PVC and PV as well, specify the `-a` switch. The backing volume on the storage will still need to be removed manually.

> **Warning:** If you remove Trident's PVC, PV and/or backing volume, you will need to reconfigure Trident from scratch if you install it again. Also, it will no longer manage any of the PVs it had provisioned.

2.2.2 Worker preparation

All of the worker nodes in the Kubernetes cluster need to be able to mount the volumes that users have provisioned for their pods.

If you are using the `ontap-nas`, `ontap-nas-economy`, `ontap-nas-flexgroup` driver for one of your backends, your workers will need the *NFS* tools. Otherwise they require the *iSCSI* tools.

Note: Recent versions of CoreOS have both installed by default.

Warning: You should always reboot your worker nodes after installing the NFS or iSCSI tools, or attaching volumes to containers may fail.

NFS

Install the following system packages:

RHEL / CentOS

```
sudo yum install -y nfs-utils
```

Ubuntu / Debian

```
sudo apt-get install -y nfs-common
```

iSCSI

RHEL / CentOS

1. Install the following system packages:

```
sudo yum install -y lsscsi iscsi-initiator-utils sg3_utils device-mapper-multipath
```

2. Enable multipathing:

```
sudo mpathconf --enable --with_multipathd y
```

3. Ensure that `iscsid` and `multipathd` are running:

```
sudo systemctl enable iscsid multipathd
sudo systemctl start iscsid multipathd
```

4. Start and enable `iscsi`:

```
sudo systemctl enable iscsi
sudo systemctl start iscsi
```

Ubuntu / Debian

1. Install the following system packages:

```
sudo apt-get install -y open-iscsi lsscsi sg3-utils multipath-tools scsitools
```

2. Enable multipathing:

```
sudo tee /etc/multipath.conf <<-'EOF'
defaults {
    user_friendly_names yes
    find_multipaths yes
}
EOF

sudo systemctl enable multipath-tools.service
sudo service multipath-tools restart
```

3. Ensure that `open-iscsi` and `multipath-tools` are enabled and running:

```
sudo systemctl status multipath-tools
sudo systemctl enable open-iscsi.service
sudo service open-iscsi start
sudo systemctl status open-iscsi
```

2.2.3 Backend configuration

A Trident backend defines the relationship between Trident and a storage system. It tells Trident how to communicate with that storage system and how Trident should provision volumes from it.

Trident will automatically offer up storage pools from backends that together match the requirements defined by a storage class.

To get started, choose the storage system type that you will be using as a backend:

Element (SolidFire)

To create and use a SolidFire backend, you will need:

- A *supported SolidFire storage system*
- Complete *SolidFire backend preparation*
- Credentials to a SolidFire cluster admin or tenant user that can manage volumes

Preparation

All of your Kubernetes worker nodes must have the appropriate iSCSI tools installed. See the *worker configuration guide* for more details.

If you're using CHAP (`UseCHAP` is *true*), no further preparation is required. It is recommended to explicitly set the `UseCHAP` option to use CHAP. Otherwise, see the *access groups guide* below.

If neither `AccessGroups` or `UseCHAP` are set then one of the following rules applies: * If the default `trident` access group is detected then access groups are used. * If no access group is detected and Kubernetes version >= 1.7 then CHAP is used.

Backend configuration options

Parameter	Description	Default
version	Always 1	
storageDriver-Name	Always "solidfire-san"	
backendName	Custom name for the storage backend	"solidfire_" + storage (iSCSI) IP address
Endpoint	MVIP for the SolidFire cluster with tenant credentials	
SVIP	Storage (iSCSI) IP address and port	
TenantName	Tenant name to use (created if not found)	
InitiatorIFace	Restrict iSCSI traffic to a specific host interface	"default"
UseCHAP	Use CHAP to authenticate iSCSI	
AccessGroups	List of Access Group IDs to use	Finds the ID of an access group named "trident"
Types	QoS specifications (see below)	
limitVolume-Size	Fail provisioning if requested volume size is above this value	"" (not enforced by default)

Example configuration

```
{
    "version": 1,
    "storageDriverName": "solidfire-san",
    "Endpoint": "https://<user>:<password>@<mvip>/json-rpc/8.0",
    "SVIP": "<svip>:3260",
    "TenantName": "<tenant>",
    "UseCHAP": true,
    "Types": [{"Type": "Bronze", "Qos": {"minIOPS": 1000, "maxIOPS": 2000, "burstIOPS
↪": 4000}},
              {"Type": "Silver", "Qos": {"minIOPS": 4000, "maxIOPS": 6000, "burstIOPS
↪": 8000}},
              {"Type": "Gold", "Qos": {"minIOPS": 6000, "maxIOPS": 8000, "burstIOPS":␣
↪10000}}]
}
```

In this case we're using CHAP authentication and modeling three volume types with specific QoS guarantees. Most likely you would then define storage classes to consume each of these using the `IOPS` storage class parameter.

Using access groups

Note: Ignore this section if you are using CHAP, which we recommend to simplify management and avoid the scaling limit described below.

Trident can use volume access groups to control access to the volumes that it provisions. If CHAP is disabled it expects to find an access group called `trident` unless one or more access group IDs are specified in the configuration.

While Trident associates new volumes with the configured access group(s), it does not create or otherwise manage access groups themselves. The access group(s) must exist before the storage backend is added to Trident, and they

need to contain the iSCSI IQNs from every node in the Kubernetes cluster that could potentially mount the volumes provisioned by that backend. In most installations that's every worker node in the cluster.

For Kubernetes clusters with more than 64 nodes, you will need to use multiple access groups. Each access group may contain up to 64 IQNs, and each volume can belong to 4 access groups. With the maximum 4 access groups configured, any node in a cluster up to 256 nodes in size will be able to access any volume.

If you're modifying the configuration from one that is using the default `trident` access group to one that uses others as well, include the ID for the `trident` access group in the list.

ONTAP (AFF/FAS/Select/Cloud)

To create and use an ONTAP backend, you will need:

- A *supported ONTAP storage system*
- Choose the *ONTAP storage driver* that you want to use
- Complete *ONTAP backend preparation* for the driver of your choice
- Credentials to an ONTAP SVM with *appropriate access*

Choosing a driver

Driver	Protocol
ontap-nas	NFS
ontap-nas-economy	NFS
ontap-nas-flexgroup	NFS
ontap-san	iSCSI

The `ontap-nas` and `ontap-san` drivers create an ONTAP FlexVol for each volume. ONTAP supports up to 1000 FlexVols per cluster node with a cluster maximum of 12,000 FlexVols. If your persistent volume requirements fit within that limitation, those drivers are the preferred solution due to the granular data management capabilities they afford.

If you need more persistent volumes than may be accommodated by the FlexVol limits, choose the `ontap-nas-economy` driver, which creates volumes as ONTAP Qtrees within a pool of automatically managed FlexVols. Qtrees offer far greater scaling, up to 100,000 per cluster node and 2,400,000 per cluster, at the expense of granular data management features.

Choose the ontap-nas-flexgroup driver to increase parallelism to a single volume that can grow into the petabyte range with billions of files. Some ideal use cases for FlexGroups include AI/ML/DL, big data and analytics, software builds, streaming, file repositories, etc. Trident uses all aggregates assigned to an SVM when provisioning a FlexGroup Volume. FlexGroup support in Trident also has the following considerations:

- Requires ONTAP version 9.2 or greater.
- As of this writing, FlexGroups only support NFSv3 (required to set `mountOptions: ["nfsvers=3"]` in the Kubernetes storage class).
- Recommended to enable the 64-bit NFSv3 identifiers for the SVM.
- The minimum recommended FlexGroup size is 100GB.
- Cloning is not supported for FlexGroup Volumes.

For information regarding FlexGroups and workloads that are appropriate for FlexGroups see the NetApp FlexGroup Volume - Best Practices and Implementation Guide.

Remember that you can also run more than one driver, and create storage classes that point to one or the other. For example, you could configure a *Gold* class that uses the `ontap-nas` driver and a *Bronze* class that uses the `ontap-nas-economy` one.

Preparation

For all ONTAP backends, Trident requires at least one aggregate assigned to the SVM.

ontap-nas, ontap-nas-economy, ontap-nas-flexgroups

All of your Kubernetes worker nodes must have the appropriate NFS tools installed. See the *worker configuration guide* for more details.

Trident uses NFS export policies to control access to the volumes that it provisions. It uses the `default` export policy unless a different export policy name is specified in the configuration.

While Trident associates new volumes (or qtrees) with the configured export policy, it does not create or otherwise manage export policies themselves. The export policy must exist before the storage backend is added to Trident, and it needs to be configured to allow access to every worker node in the Kubernetes cluster.

If the export policy is locked down to specific hosts, it will need to be updated when new nodes are added to the cluster, and that access should be removed when nodes are removed as well.

ontap-san

All of your Kubernetes worker nodes must have the appropriate iSCSI tools installed. See the *worker configuration guide* for more details.

Trident uses igroups to control access to the volumes (LUNs) that it provisions. It expects to find an igroup called `trident` unless a different igroup name is specified in the configuration.

While Trident associates new LUNs with the configured igroup, it does not create or otherwise manage igroups themselves. The igroup must exist before the storage backend is added to Trident, and it needs to contain the iSCSI IQNs from every worker node in the Kubernetes cluster.

The igroup needs to be updated when new nodes are added to the cluster, and they should be removed when nodes are removed as well.

Backend configuration options

Parameter	Description	Default
version	Always 1	
storageDriverName	"ontap-nas", "ontap-nas-economy", "ontap-nas-flexgroup", or "ontap-san"	
backendName	Custom name for the storage backend	Driver name + "_" + dataLIF
managementLIF	IP address of a cluster or SVM management LIF	"10.0.0.1"
dataLIF	IP address of protocol LIF	Derived by the SVM unless specified
svm	Storage virtual machine to use	Derived if an SVM managementLIF is specified
igroupName	Name of the igroup for SAN volumes to use	"trident"
username	Username to connect to the cluster/SVM	
password	Password to connect to the cluster/SVM	
storagePrefix	Prefix used when provisioning new volumes in the SVM	"trident"
limitAggregateUsage	Fail provisioning if usage is above this percentage	"" (not enforced by default)
limitVolumeSize	Fail provisioning if requested volume size is above this value	"" (not enforced by default)
nfsMountOptions	Comma-separated list of NFS mount options (except ontap-san)	""

A fully-qualified domain name (FQDN) can be specified for the managementLIF option. For the ontap-nas* drivers only, a FQDN may also be specified for the dataLIF option, in which case the FQDN will be used for the NFS mount operations. For the ontap-san driver, the default is to use all data LIF IPs from the SVM and to use iSCSI multipath. Specifying an IP address for the dataLIF for the ontap-san driver forces the driver to disable multipath and use only the specified address. For the ontap-nas-economy driver, the limitVolumeSize option will also restrict the maximum size of the volumes it manages for qtrees.

The nfsMountOptions parameter applies to all ONTAP drivers except ontap-san. The mount options for Kubernetes persistent volumes are normally specified in storage classes, but if no mount options are specified in a storage class, Trident will fall back to using the mount options specified in the storage backend's config file. If no mount options are specified in either the storage class or the config file, then Trident will not set any mount options on an associated persistent volume.

You can control how each volume is provisioned by default using these options in a special section of the configuration. For an example, see the configuration examples below.

Parameter	Description	Default
spaceReserve	Space reservation mode; "none" (thin) or "volume" (thick)	"none"
snapshotPolicy	Snapshot policy to use	"none"
snapshotReserve	Percentage of volume reserved for snapshots	"0" if snapshotPolicy is "none", else ""
splitOnClone	Split a clone from its parent upon creation	"false"
encryption	Enable NetApp volume encryption	"false"
unixPermissions	ontap-nas* only: mode for new volumes	"777"
snapshotDir	ontap-nas* only: access to the .snapshot directory	"false"
exportPolicy	ontap-nas* only: export policy to use	"default"
securityStyle	ontap-nas* only: security style for new volumes	"unix"

Example configuration

NFS Example for ontap-nas driver

```
{
    "version": 1,
    "storageDriverName": "ontap-nas",
    "managementLIF": "10.0.0.1",
    "dataLIF": "10.0.0.2",
    "svm": "svm_nfs",
    "username": "vsadmin",
    "password": "secret",
    "limitAggregateUsage": "80%",
    "limitVolumeSize": "50Gi",
    "nfsMountOptions": "nfsvers=4",
    "defaults": {
      "spaceReserve": "volume",
      "exportPolicy": "myk8scluster",
      "snapshotPolicy": "default",
      "snapshotReserve": "10"
    }
}
```

NFS Example for ontap-nas-flexgroup driver

```
{
    "version": 1,
    "storageDriverName": "ontap-nas-flexgroup",
    "managementLIF": "10.0.0.1",
    "dataLIF": "10.0.0.2",
    "svm": "svm_nfs",
    "username": "vsadmin",
    "password": "secret",
    "defaults": {
      "size": "100G",
      "spaceReserve": "volume",
      "exportPolicy": "myk8scluster"
    }
}
```

NFS Example for ontap-nas-economy driver

```
{
    "version": 1,
    "storageDriverName": "ontap-nas-economy",
    "managementLIF": "10.0.0.1",
    "dataLIF": "10.0.0.2",
    "svm": "svm_nfs",
    "username": "vsadmin",
    "password": "secret"
}
```

iSCSI Example for ontap-san driver

```
{
    "version": 1,
    "storageDriverName": "ontap-san",
```

(continues on next page)

(continued from previous page)

```
    "managementLIF": "10.0.0.1",
    "dataLIF": "10.0.0.3",
    "svm": "svm_iscsi",
    "igroupName": "trident",
    "username": "vsadmin",
    "password": "secret"
}
```

User permissions

Trident expects to be run as either an ONTAP or SVM administrator, typically using the `admin` cluster user or a `vsadmin` SVM user, or a user with a different name that has the same role.

Note: If you use the "limitAggregateUsage" option, cluster admin permissions are required.

While it is possible to create a more restrictive role within ONTAP that a Trident driver can use, we don't recommend it. Most new releases of Trident will call additional APIs that would have to be accounted for, making upgrades difficult and error-prone.

SANtricity (E-Series)

To create and use an E-Series backend, you will need:

- A *supported E-Series storage system*
- Complete *E-Series backend preparation*
- Credentials to the E-Series storage system

Preparation

All of your Kubernetes worker nodes must have the appropriate iSCSI tools installed. See the *worker configuration guide* for more details.

Trident uses host groups to control access to the volumes (LUNs) that it provisions. It expects to find a host group called `trident` unless a different host group name is specified in the configuration.

While Trident associates new volumes with the configured host group, it does not create or otherwise manage host groups themselves. The host group must exist before the storage backend is added to Trident, and it needs to contain a host definition with an iSCSI IQN for every worker node in the Kubernetes cluster.

Backend configuration options

Parameter	Description	Default
version	Always 1	
storageDriverName	Always "eseries-iscsi"	
backendName	Custom name for the storage backend	"eseries_" + hostDataIP
webProxyHostname	Hostname or IP address of the web services proxy	
webProxyPort	Port number of the web services proxy	80 for HTTP, 443 for HTTPS
webProxyUseHTTP	Use HTTP instead of HTTPS to communicate to the proxy	false
webProxyVerifyTLS	Verify certificate chain and hostname	false
username	Username for the web services proxy	
password	Password for the web services proxy	
controllerA	IP address for controller A	
controllerB	IP address for controller B	
passwordArray	Password for the storage array, if set	""
hostDataIP	Host iSCSI IP address	
poolNameSearchPattern	Regular expression for matching available storage pools	".+" (all)
hostType	E-Series Host types created by the driver	"linux_dm_mp"
accessGroupName	E-Series Host Group used by the driver	"trident"
limitVolumeSize	Fail provisioning if requested volume size is above this value	"" (not enforced by default)

Example configuration

```
{
  "version": 1,
  "storageDriverName": "eseries-iscsi",
  "webProxyHostname": "localhost",
  "webProxyPort": "8443",
  "webProxyUseHTTP": false,
  "webProxyVerifyTLS": true,
  "username": "rw",
  "password": "rw",
  "controllerA": "10.0.0.5",
  "controllerB": "10.0.0.6",
  "passwordArray": "",
  "hostDataIP": "10.0.0.101"
}
```

2.2.4 Managing backends

Creating a backend configuration

We have an entire *backend configuration* guide to help you with this.

Creating a backend

Once you have a *backend configuration* file, run:

```
tridentctl create backend -f <backend-file>
```

If backend creation fails, something was wrong with the backend configuration. You can view the logs to determine the cause by running:

```
tridentctl logs
```

Once you identify and correct the problem with the configuration file you can simply run the create command again.

Deleting a backend

Note: If Trident has provisioned volumes from this backend that still exist, deleting the backend will prevent new volumes from being provisioned by it but the backend will continue to exist and Trident will continue to manage those volumes until they are deleted.

To delete a backend from Trident, run:

```
# Retrieve the backend name
tridentctl get backend

tridentctl delete backend <backend-name>
```

Viewing the existing backends

To view the backends that Trident knows about, run:

```
# Summary
tridentctl get backend

# Full details
tridentctl get backend -o json
```

Identifying the storage classes that will use a backend

This is an example of the kind of questions you can answer with the JSON that `tridentctl` outputs for Trident backend objects. This uses the `jq` utility, which you may need to install first.

```
tridentctl get backend -o json | jq '[.items[] | {backend: .name, storageClasses: [.
↪storage[].storageClasses]|unique}]'
```

Updating a backend

Once you have a new *backend configuration* file, run:

```
tridentctl update backend <backend-name> -f <backend-file>
```

If backend update fails, something was wrong with the backend configuration or you attempted an invalid update. You can view the logs to determine the cause by running:

```
tridentctl logs
```

Once you identify and correct the problem with the configuration file you can simply run the update command again.

2.2.5 Managing storage classes

Designing a storage class

The *StorageClass concept guide* will help you understand what they do and how you configure them.

Creating a storage class

Once you have a storage class file, run:

```
kubectl create -f <storage-class-file>
```

Deleting a storage class

To delete a storage class from Kubernetes, run:

```
kubectl delete storageclass <storage-class>
```

Any persistent volumes that were created through this storage class will remain untouched, and Trident will continue to manage them.

Viewing the existing storage classes

```
# Kubernetes storage classes
kubectl get storageclass

# Kubernetes storage class detail
kubectl get storageclass <storage-class> -o json

# Trident's synchronized storage classes
tridentctl get storageclass

# Trident's synchronized storage class detail
tridentctl get storageclass <storage-class> -o json
```

Setting a default storage class

Kubernetes v1.6 added the ability to set a default storage class. This is the storage class that will be used to provision a PV if a user does not specify one in a PVC.

You can define a default storage class by setting the annotation `storageclass.kubernetes.io/is-default-class` to `true` in the storage class definition. According to the specification, any other value or absence of the annotation is interpreted as `false`.

It is possible to configure an existing storage class to be the default storage class by using the following command:

```
kubectl patch storageclass <storage-class-name> -p '{"metadata": {"annotations":{
↪"storageclass.kubernetes.io/is-default-class":"true"}}}'
```

Similarly, you can remove the default storage class annotation by using the following command:

```
kubectl patch storageclass <storage-class-name> -p '{"metadata": {"annotations":{
↪"storageclass.kubernetes.io/is-default-class":"false"}}}'
```

There are also examples in the Trident installer bundle that include this annotation.

Note: You should only have one default storage class in your cluster at any given time. Kubernetes does not technically prevent you from having more than one, but it will behave as if there is no default storage class at all.

Identifying the Trident backends that a storage class will use

This is an example of the kind of questions you can answer with the JSON that `tridentctl` outputs for Trident backend objects. This uses the `jq` utility, which you may need to install first.

```
tridentctl get storageclass -o json | jq  '[.items[] | {storageClass: .Config.name,␣
↪backends: [.storage]|unique}]'
```

2.2.6 Managing volumes

Resizing an NFS volume

Starting with v18.10, Trident supports volume resize for NFS PVs. More specifically, PVs provisioned on `ontap-nas`, `ontap-nas-economy`, and `ontap-nas-flexgroup` backends can be expanded.

Resizing Persistent Volumes using Kubernetes blog post describes the workflows involved in resizing a PV. Volume resize was introduced in Kubernetes v1.8 as an alpha feature and was promoted to beta in v1.11, which means this feature is enabled by default starting with Kubernetes v1.11.

Because NFS PV resize is not supported by Kubernetes, and is implemented by the Trident orchestrator externally, Kubernetes admission controller may reject PVC size updates for in-tree volume plugins that don't support resize (e.g., NFS). The Trident team has changed Kubernetes to allow such changes starting with Kubernetes 1.12. Therefore, we recommend using this feature with Kubernetes 1.12 or later as it would just work.

While we recommend using Kubernetes 1.12 or later, it is still possible to resize NFS PVs with earlier versions of Kubernetes that support resize. This is done by disabling the `PersistentVolumeClaimResize` admission plugin when the Kubernetes API server is started:

```
kube-apiserver --disable-admission-plugins=PersistentVolumeClaimResize
```

With Kubernetes 1.8-1.10 that offer this feature as alpha, the `ExpandPersistentVolumes` Feature Gate should also be turned on:

```
kube-apiserver --feature-gates=ExpandPersistentVolumes=true --disable-admission-
↪plugins=PersistentVolumeClaimResize
```

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: netapp.io/trident
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 via `kubectl edit storageclass` to allow volume expansion.

Next, we create a PVC using this storage class:

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

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

```
$ kubectl get pvc
NAME                    STATUS    VOLUME                                     CAPACITY
↪ACCESS MODES   STORAGECLASS        AGE
ontapnas20mb            Bound     default-ontapnas20mb-c1bd7                 20Mi
↪RWO            ontapnas            14s

$ kubectl get pv default-ontapnas20mb-c1bd7
NAME                          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
↪CLAIM                   STORAGECLASS       REASON    AGE
default-ontapnas20mb-c1bd7   20Mi       RWO             Delete           Bound
↪default/ontapnas20mb   ontapnas                      1m
```

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

```
$ kubectl edit pvc ontapnas20mb
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file
↪will be
# reopened with the relevant failures.
#
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    pv.kubernetes.io/bind-completed: "yes"
    pv.kubernetes.io/bound-by-controller: "yes"
```

(continues on next page)

(continued from previous page)

```
    volume.beta.kubernetes.io/storage-provisioner: netapp.io/trident
  creationTimestamp: 2018-08-21T18:26:44Z
  finalizers:
  - kubernetes.io/pvc-protection
  name: ontapnas20mb
  namespace: default
  resourceVersion: "1958015"
  selfLink: /api/v1/namespaces/default/persistentvolumeclaims/ontapnas20mb
  uid: c1bd7fa5-a56f-11e8-b8d7-fa163e59eaab
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
...
```

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

```
$ kubectl get pvc ontapnas20mb
NAME           STATUS    VOLUME                         CAPACITY   ACCESS MODES  ␣
↪STORAGECLASS       AGE
ontapnas20mb   Bound     default-ontapnas20mb-c1bd7   1Gi        RWO           ␣
↪ontapnas           6m

$ kubectl get pv default-ontapnas20mb-c1bd7
NAME                         CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS    ␣
↪CLAIM                   STORAGECLASS       REASON    AGE
default-ontapnas20mb-c1bd7   1Gi        RWO            Delete           Bound     ␣
↪default/ontapnas20mb   ontapnas             6m

$ tridentctl get volume default-ontapnas20mb-c1bd7 -n trident
+-----------------------------+---------+------------------+----------+----------------
↪--------+--------------+
|             NAME            |  SIZE   |  STORAGE CLASS   | PROTOCOL |        BACKEND␣
↪        |     POOL     |
+-----------------------------+---------+------------------+----------+----------------
↪--------+--------------+
| default-ontapnas20mb-c1bd7 | 1.0 GiB | ontapnas         | file     | ontapnas_10.63.
↪171.111 | VICE08_aggr1 |
+-----------------------------+---------+------------------+----------+----------------
↪--------+--------------+
```

2.3 Production considerations

2.3.1 etcd

Trident's use of etcd

Trident uses etcd to maintain state for the *objects* that it manages.

By default, Trident deploys an etcd container as part of the Trident pod. This is a single node etcd cluster managed by Trident that's backed by a highly reliable volume from a NetApp storage system. This is perfectly acceptable for production.

Using an external etcd cluster

In some cases there may already be a production etcd cluster available that you would like Trident to use instead, or you would like to build one.

Note: Kubernetes itself uses an etcd cluster for its objects. While it's technically possible to use that cluster to store Trident's state as well, it is highly discouraged by the Kubernetes community.

Beginning with Trident 18.01, this is a supported configuration provided that the etcd cluster is using v3. It is also highly recommend that you encrypt communication to the remote cluster with TLS.

The instructions in this section cover both the case where you are deploying Trident for the first time with an external etcd cluster and the case where you already have a Trident deployment that uses the local etcd container and you want to move to an external etcd cluster without losing any state maintained by Trident.

Step 1: Bring down Trident

First, make sure that you have started Trident successfully at least once with version 18.01 or above. Previous versions of Trident used etcdv2, and Trident needs to start once with a more recent version to automatically upgrade its state to the etcdv3 format.

Now we need to bring down Trident so that no new state is written to etcd. This is most easily done by running the uninstall script, which retains all state by default.

The uninstall script is located in the Trident installer bundle that you downloaded to install Trident.

```
trident-installer$ ./uninstall_trident.sh -n <namespace>
```

Step 2: Copy scripts and the deployment file

As part of this step, we copy three files to the root of the Trident installer bundle directory:

```
trident-installer$ ls extras/external-etcd/trident/
etcdcopy-job.yaml  install_trident_external_etcd.sh  trident-deployment-external-etcd.
↪yaml
trident-installer$ cp extras/external-etcd/trident/* .
```

`etcdcopy-job.yaml` contains the definition for an application that copies etcd data from one endpoint to another. If you are setting up a new Trident instance with an external cluster for the first time, you can ignore this file and Step 3.

`trident-deployment-external-etcd.yaml` contains the Deployment definition for the Trident instance that is configured to work with an external cluster. This file is used by the `install_trident_external_etcd.sh` script.

As the contents of `trident-deployment-external-etcd.yaml` and `install_trident_external_etcd.sh` suggest, the install process is much simpler with an external etcd cluster as there is no need to run Trident launcher to provision a volume, PVC, and PV for the Trident deployment.

Step 3: Copy etcd data from the local endpoint to the remote endpoint

Once you make sure that Trident is not running as instructed in Step 1, configure `etcdcopy-job.yaml` with the information about the destination cluster. In this example, we are copying data from the local etcd instance used by the terminated Trident deployment to the remote cluster.

`etcdcopy-job.yaml` makes reference to the Kubernetes Secret `etcd-client-tls`, which was created automatically if you installed the sample etcd cluster. If you already have a production etcd cluster set up, you need to

generate the Secret yourself and adjust the parameters taken by the `etcd-copy` container in `etcdcopy-job.yaml`.

For example, `etcdcopy-job.yaml` is based on a Secret that was created using the following command:

```
trident-installer/extras/external-etcd$ kubectl --namespace=trident create secret␣
↪generic etcd-client-tls --from-file=etcd-client-ca.crt=./certs/ca.pem --from-
↪file=etcd-client.crt=./certs/client.pem --from-file=etcd-client.key=./certs/client-
↪key.pem
```

Based on how you set up your external cluster and how you name the files that make up the Secret, you may have to modify `etcdv3_dest` arguments in `etcdcopy-job.yaml` as well:

```
- -etcdv3_dest
- https://trident-etcd-client:2379
- -etcdv3_dest_cacert
- /root/certs/etcd-client-ca.crt
- -etcdv3_dest_cert
- /root/certs/etcd-client.crt
- -etcdv3_dest_key
- /root/certs/etcd-client.key
```

Once `etcdcopy-job.yaml` is configured properly, you can start migrating data between the two etcd endpoints:

```
trident-installer$ kubectl create -f etcdcopy-job.yaml
job "etcd-copy" created
trident-installer$ kubectl get pod -aw
NAME                              READY     STATUS      RESTARTS   AGE
etcd-copy-fzhqm                   1/2       Completed   0          14s
etcd-operator-3986959281-782hx    1/1       Running     0          1d
etcdctl                           1/1       Running     0          1d
trident-etcd-0000                 1/1       Running     0          1d
trident-installer$ kubectl logs etcd-copy-fzhqm -c etcd-copy
time="2017-11-03T14:36:35Z" level=debug msg="Read key from the source." key="/trident/
↪v1/backend/solidfire_10.250.118.144"
time="2017-11-03T14:36:35Z" level=debug msg="Wrote key to the destination." key="/
↪trident/v1/backend/solidfire_10.250.118.144"
time="2017-11-03T14:36:35Z" level=debug msg="Read key from the source." key="/trident/
↪v1/storageclass/solidfire"
time="2017-11-03T14:36:35Z" level=debug msg="Wrote key to the destination." key="/
↪trident/v1/storageclass/solidfire"
trident-installer$ kubectl delete -f etcdcopy-job.yaml
job "etcd-copy" deleted
```

The logs for *etcd-copy* should indicate that Job has successfully copied Trident's state to the remote etcd cluster.

Step 4: Install Trident with an external etcd cluster

Prior to running the install script, please adjust `trident-deployment-external-etcd.yaml` to reflect your setup. More specifically, you may need to change the etcdv3 endpoint and Secret if you did not rely on the instructions on this page to set up your etcd cluster.

```
trident-installer$ ./install_trident_external_etcd.sh -n trident
```

That's it! Trident is now up and running against an external etcd cluster. You should now be able to run *tridentctl* and see all of the same configuration you had before.

Building your own etcd cluster

We needed to be able to easily create etcd clusters with RBAC and TLS enabled for testing purposes. We think that the tools we built to do that are also a useful way to help others understand how to do that.

This provides a reference to show how Trident operates with an external etcd cluster, and it should be generic enough to use for applications other than Trident.

These instructions use the etcd operator and are based on the information found in Cluster Spec, Cluster TLS Guide, etcd Client Service, Operator RBAC Setup, and Generating Self-signed Certificates.

Installing

The Trident installer bundle includes a set of scripts and configuration files to set up an external cluster. These files can be found under `trident-installer/extras/external-etcd/`.

To install the etcd cluster in namespace `trident`, run the following command:

```
trident-installer$ cd extras/external-etcd/
trident-installer/extras/external-etcd$ ./install_etcd.sh -n trident
Installer assumes you have deployed Kubernetes. If this is an OpenShift deployment,␣
↪make sure 'oc' is in the $PATH.
cfssl and cfssljson have already been downloaded.
serviceaccount "etcd-operator" created
clusterrole "etcd-operator" created
clusterrolebinding "etcd-operator" created
deployment "etcd-operator" created
secret "etcd-client-tls" created
secret "etcd-server-tls" created
secret "etcd-peer-tls" created
etcdcluster "trident-etcd" created
```

The above script creates a few Kubernetes objects, including the following:

```
trident-installer/extras/external-etcd$ kubectl get pod
NAME                              READY     STATUS    RESTARTS   AGE
etcd-operator-3986959281-m0481   1/1       Running   0          1m
trident-etcd-0000                 1/1       Running   0          20s
trident-installer/extras/external-etcd$ kubectl get service
NAME                  CLUSTER-IP    EXTERNAL-IP   PORT(S)             AGE
trident-etcd          None          <none>        2379/TCP,2380/TCP   1m
trident-etcd-client   10.99.21.44   <none>        2379/TCP            1m
trident-installer/extras/external-etcd$ kubectl get secret
NAME                       TYPE                                  DATA      AGE
default-token-ql7s3        kubernetes.io/service-account-token   3         72d
etcd-client-tls            Opaque                                3         1m
etcd-operator-token-nsh2n  kubernetes.io/service-account-token   3         1m
etcd-peer-tls              Opaque                                3         1m
etcd-server-tls            Opaque                                3         1m
```

The Kubernetes Secrets shown above are constructed using the CA, certificates, and private keys generated by the installer script:

```
trident-installer/extras/external-etcd$ ls certs/
ca-config.json  ca-key.pem  client-csr.json  gen-ca.sh      gen-server.sh  peer-key.
↪pem  server-csr.json
```

(continues on next page)

(continued from previous page)

```
ca.csr          ca.pem      client-key.pem   gen-client.sh  peer.csr       peer.pem
↪   server-key.pem
ca-csr.json     client.csr  client.pem       gen-peer.sh    peer-csr.json  server.csr
↪   server.pem
```

For more information about the Secrets used by the operator, please see Cluster TLS Guide and Generating Self-signed Certificates.

Testing

To verify the cluster we brought up in the previous step is working properly, we can run the following commands:

```
trident-installer/extras/external-etcd$ kubectl create -f kubernetes-yaml/etcdctl-pod.
↪yaml
trident-installer/extras/external-etcd$ kubectl exec etcdctl -- etcdctl --
↪endpoints=https://trident-etcd-client:2379 --cert=/root/certs/etcd-client.crt --
↪key=/root/certs/etcd-client.key --cacert=/root/certs/etcd-client-ca.crt member list
↪-w table
trident-installer/extras/external-etcd$ kubectl exec etcdctl -- etcdctl --
↪endpoints=https://trident-etcd-client:2379 --cert=/root/certs/etcd-client.crt --
↪key=/root/certs/etcd-client.key --cacert=/root/certs/etcd-client-ca.crt put foo bar
trident-installer/extras/external-etcd$ kubectl exec etcdctl -- etcdctl --
↪endpoints=https://trident-etcd-client:2379 --cert=/root/certs/etcd-client.crt --
↪key=/root/certs/etcd-client.key --cacert=/root/certs/etcd-client-ca.crt get foo
trident-installer/extras/external-etcd$ kubectl exec etcdctl -- etcdctl --
↪endpoints=https://trident-etcd-client:2379 --cert=/root/certs/etcd-client.crt --
↪key=/root/certs/etcd-client.key --cacert=/root/certs/etcd-client-ca.crt del foo
trident-installer/extras/external-etcd$ kubectl delete -f kubernetes-yaml/etcdctl-pod.
↪yaml
```

The above commands invoke the `etcdctl` binary inside the etcdctl pod to interact with the etcd cluster. Please see `kubernetes-yaml/etcdctl-pod.yaml` to understand how client credentials are supplied using the `etcd-client-tls` Secret to the etcdctl pod. It is important to note that etcd operator requires a working kube-dns pod as it relies on a Kubernetes Service to communicate with the etcd cluster.

Uninstalling

To uninstall the etcd cluster in namespace *trident*, run the following:

```
trident-installer/extras/external-etcd$ ./uninstall_etcd.sh -n trident
```

2.4 Concepts

2.4.1 Kubernetes and Trident objects

Both Kubernetes and Trident are designed to be interacted with through REST APIs by reading and writing resource objects.

Object overview

There are several different resource objects in play here, some that are managed through Kubernetes and others that are managed through Trident, that dictate the relationship between Kubernetes and Trident, Trident and storage, and Kubernetes and storage.

Perhaps the easiest way to understand these 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 *Trident 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* will mount that *PersistentVolume* on any host that it runs on.

Throughout the rest of this guide, we will describe the different Trident objects and details about how Trident crafts the Kubernetes PersistentVolume object for storage that it provisions.

For further reading about the 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, eseries-iscsi
trident.netapp.io/cloneFromPVC	cloneSourceVolume	ontap-nas, ontap-san, solidfire-san
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

If the created PV has the `Delete` reclaim policy, Trident will delete both the PV and the backing volume when the PV becomes released (i.e., when the user deletes the PVC). Should the delete action fail, Trident will mark the PV as

such and periodically retry 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 will not cause Trident to delete the backing volume; it must be removed manually via the REST API (i.e., `tridentctl`).

One novel aspect of Trident is that users can provision new volumes by cloning existing volumes. Trident enables this functionality via the PVC annotation `trident.netapp.io/cloneFromPVC`. For example, if a user already has a PVC called `mysql`, she can create a new PVC called `mysqlclone` by referring to the `mysql` PVC: `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. A few points worth considering are the following: (1) We recommend cloning an idle volume, (2) a PVC and its clone must be in the same Kubernetes namespace and have the same storage class, and (3) with `ontap-\*` 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; 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.

`sample-input/pvc-basic.yaml`, `sample-input/pvc-basic-clone.yaml`, and `sample-input/pvc-full.yaml` contain examples of PVC definitions for use with Trident. See *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's been made available to the Kubernetes cluster. They have a lifecycle that's independent of the pod that uses it.

Note: 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 a user creates a PVC that refers to a Trident-based `StorageClass`, Trident will provision a new volume using the corresponding storage class and register 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 may 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 that will be used and defines that set of properties in terms the provisioner understands.

It is one of two objects that need to be created and managed by you, the administrator. The other is the *Trident Backend object*.

A Kubernetes StorageClass object that uses Trident looks like this:

```
apiVersion: storage.k8s.io/v1beta1
kind: StorageClass
metadata:
  name: <Name>
provisioner: netapp.io/trident
mountOptions: <Mount Options>
parameters:
  <Trident Parameters>
```

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
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 two groups:

1. 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	string	hdd, hybrid, ssd	Pool contains media of this type; hybrid means both	Media type specified	All drivers
provisioningType	string	thin, thick	Pool supports this provisioning method	Provisioning method specified	thick: all but solidfire-san, thin: all but eseries-iscsi
backendType	string	ontap-nas, ontap-nas-economy, ontap-nas-flexgroup, ontap-san, solidfire-san, eseries-iscsi	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

In most cases, the values requested will directly influence provisioning; for instance, requesting thick provisioning will result in a thickly provisioned volume. However, a SolidFire storage pool will use 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 will be able to use `attributes` alone to model the qualities of the storage you need to satisfy the needs of a particular class. Trident will automatically discover and select 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 manually.

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

The `additionalStoragePools` parameter is used to extend the set of pools that Trident will use for provisioning, regardless of any pools selected by the `attributes` and `storagePools` parameters.

The `excludeStoragePools` parameter is used to filter the set of pools that Trident will use for provisioning and will remove 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 will 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.

2. 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.

Attribute	Type	Values	Description	Relevant Drivers	Kubernetes Version
fsType	string	ext4, ext3, xfs, etc.	The file system type for block volumes	solidfire-san, ontap-san, eseries-iscsi	All

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 will cause the corresponding Trident storage class to be deleted as well.

Trident StorageClass objects

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

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

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.

One can create storage class configurations to directly define storage classes via the *REST API*. However, for Kubernetes deployments, we expect them to be created as a side-effect of 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 will need to create and manage yourself. The other is the *Kubernetes StorageClass object* below.

For more information about how to construct these objects, visit the *backend configuration* guide.

Trident StoragePool objects

Storage pools represent the distinct locations available for provisioning on each backend. For ONTAP, these correspond to aggregates in SVMs; for SolidFire, these correspond to admin-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. *View your backends* to see the storage pools associated with them.

Trident Volume objects

Note: With Kubernetes, these objects are managed automatically and should not be manipulated by hand. You can view them to see what Trident provisioned, however.

Volumes are the basic unit of provisioning, comprising backend endpoints such as NFS shares and iSCSI LUNs. In Kubernetes, these correspond directly to PersistentVolumes. Each volume must be created with a storage class, which determines where that volume can be provisioned, along with a size.

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
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
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

As mentioned, 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 configurationd – 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>`), and for SolidFire, it replaces underscores with hyphens. For E-Series, which imposes a 30-character limit on all object names, Trident generates a random string for the internal name of each volume on the array.

One can use volume configurations to directly provision volumes via the *REST API*, but in Kubernetes deployments we expect most users to use the standard Kubernetes PersistentVolumeClaim method. Trident will create this volume object automatically as part of the provisioning process in that case.

2.4.2 How does provisioning work?

Provisioning in Trident has two primary phases. The first of these associates a storage class with the set of suitable backend storage pools and occurs as a necessary preparation before provisioning. The second encompasses the volume creation itself and requires choosing a storage pool from those associated with the pending volume's storage class. This section explains both of these phases and the considerations involved in them, so that users can better understand how Trident handles their storage.

Associating backend storage pools with a storage class relies on both the storage class's requested attributes and its `storagePools`, `additionalStoragePools`, and `excludeStoragePools` lists. When a user creates a storage class, Trident compares the attributes and pools offered by each of its backends to those requested by the storage class. If a storage pool's attributes and name match all of the requested attributes and pool names, Trident adds that storage pool to the set of suitable storage pools for that storage class. In addition, Trident adds all storage pools listed in the `additionalStoragePools` list to that set, even if their attributes do not fulfill all or any of the storage class's requested attributes. Use the `excludeStoragePools` list to override and remove storage pools from use for a storage class. Trident performs a similar process every time a user adds a new backend, checking whether its storage pools satisfy those of the existing storage classes and removing any that have been marked as excluded.

Trident then uses the associations between storage classes and storage pools to determine where to provision volumes. When a user creates a volume, Trident first gets the set of storage pools for that volume's storage class, and, if the user specifies a protocol for the volume, it removes those storage pools that cannot provide the requested protocol (a SolidFire backend cannot provide a file-based volume while an ONTAP NAS backend cannot provide a block-based volume, for instance). Trident randomizes the order of this resulting set, to facilitate an even distribution of volumes, and then iterates through it, attempting to provision the volume on each storage pool in turn. If it succeeds on one, it returns successfully, logging any failures encountered in the process. Trident returns a failure if and only if it fails to provision on **all** the storage pools available for the requested storage class and protocol.

2.5 Known issues

Trident is in an early stage of development, and thus, there are several outstanding issues to be aware of when using it:

- Due to known issues in Kubernetes 1.5 (or OpenShift 3.5) and earlier, use of iSCSI with ONTAP or E-Series in production deployments is not recommended. See Kubernetes issues #40941, #41041 and #39202. ONTAP NAS and SolidFire are unaffected. These issues are fixed in Kubernetes 1.6 (and OpenShift 3.6).
- Although we provide a deployment for Trident, it should never be scaled beyond a single replica. Similarly, only one instance of Trident should be run per Kubernetes cluster. Trident cannot communicate with other instances and cannot discover other volumes that they have created, which will lead to unexpected and incorrect behavior if more than one instance runs within a cluster.
- Volumes and storage classes created in the REST API will not have corresponding objects (PVCs or `StorageClasses`) created in Kubernetes; however, storage classes created via `tridentctl` or the REST API will be usable by PVCs created in Kubernetes.

- If Trident-based `StorageClass` objects are deleted from Kubernetes while Trident is offline, Trident will not remove the corresponding storage classes from its database when it comes back online. Any such storage classes must be deleted manually using `tridentctl` or the REST API.
- If a user deletes a PV provisioned by Trident before deleting the corresponding PVC, Trident will not automatically delete the backing volume. In this case, the user must remove the volume manually via `tridentctl` or the REST API.
- Trident will not boot unless it can successfully communicate with an etcd instance. If it loses communication with etcd during the bootstrap process, it will halt; communication must be restored before Trident will come online. Once fully booted, however, Trident is resilient to etcd outages.
- When using a backend across multiple Trident instances, it is recommended that each backend configuration file specify a different `storagePrefix` value for ONTAP backends or use a different `TenantName` for SolidFire backends. Trident cannot detect volumes that other instances of Trident have created, and attempting to create an existing volume on either ONTAP or SolidFire backends succeeds as Trident treats volume creation as an idempotent operation. Thus, if the `storagePrefix` or `TenantName` does not differ, there is a very slim chance to have name collisions for volumes created on the same backend.
- Due to a known issue in the containerized version of OpenShift, where some iSCSI initiator utilities are missing, Trident will fail to install on iSCSI storage systems. This only affects OpenShift Origin. See OpenShift issues #18880.
- ONTAP cannot concurrently provision more than one FlexGroup at a time unless the set of aggregates are unique to each provisioning request.

2.6 Troubleshooting

- If there was a failure during install, run `tridentctl logs -l all -n trident` and look for problems in the logs for the `trident-main` and `etcd` containers. Alternatively, you can use `kubectl logs` to retrieve the logs for the `trident-*********-****` pod.
- If the Trident pod fails to come up properly (e.g., when Trident pod is stuck in the `ContainerCreating` phase with fewer than 2 ready containers), running `kubectl -n trident describe deployment trident` and `kubectl -n trident describe pod trident-*********-****` can provide additional insights. Obtaining kubelet logs (e.g., via `journalctl -xeu kubelet`) can also be helpful if there is a problem in mounting the `trident` PVC (the `etcd` volume).
- If there's not enough information in the Trident logs, you can try enabling the debug mode for Trident by passing the `-d` flag to the install parameter: `./tridentctl install -d -n trident`.
- The *uninstall parameter* can help with cleaning up after a failed run. By default the script does not touch the etcd backing store, making it safe to uninstall and install again even in a running deployment.
- If Trident fails to start and the logs from Trident's etcd container report "another etcd process is running with the same data dir and holding the file lock" or similar, then you may have stale NFSv3 locks held on the ONTAP storage system. This situation may be caused by an unclean shutdown of the Kubernetes node where Trident is running. You can avoid this issue by enabling NFSv4 on your ONTAP SVM and setting `nfsMountOptions: "nfsvers=4"` in the backend.json config file used during Trident installation. Furthermore, you should use `kubectl drain` or `oc adm drain` to cleanly stop all pods on a Kubernetes node prior to powering it off.
- After a successful install, if a PVC is stuck in the `Pending` phase, running `kubectl describe pvc` can provide additional information on why Trident failed to provsion a PV for this PVC.
- If you require further assistance, please create a support bundle via `tridentctl logs -a -n trident` and send it to *NetApp Support*.

2.7 CSI Trident for Kubernetes

2.7.1 CSI overview

The container storage interface (CSI) is a standardized API for container orchestrators to manage storage plugins. A community-driven effort, CSI promises to let storage vendors write a single plugin to make their products work with multiple container orchestrators.

Kubernetes 1.10 contains beta-level support for CSI-based plugins. It uses CSI version 0.2, which has very limited capabilities, including creating/deleting/mounting/unmounting persistent volumes.

2.7.2 CSI Trident

As an independent external volume provisioner, Trident for Kubernetes offers many capabilities beyond those allowed by CSI, so it is unlikely that Trident will limit itself to CSI in the foreseeable future.

However, NetApp is participating in the community's efforts to see CSI achieve its full potential. As part of that work, we have developed a public preview version of Trident that deploys itself as a CSI plugin. The goals of shipping CSI Trident are to better inform our contributions to CSI as well as to give our customers a way to kick the tires on CSI and see where the container storage community is going.

Note: This experimental CSI driver does not support Kubernetes v1.13

Warning: CSI Trident for Kubernetes is an unsupported, alpha-level preview release for evaluation purposes only, and it should not be deployed in support of production workloads. We recommend you install CSI Trident in a sandbox cluster used primarily for testing and evaluation.

Installing CSI Trident

To install CSI Trident, follow the the extensive *deployment* guide, with just one difference.

Invoke the install command with the `--csi` switch:

```
#./tridentctl install -n trident --csi
WARN CSI Trident for Kubernetes is a technology preview and should not be installed␣
↪in production environments!
INFO Starting Trident installation.                namespace=trident
INFO Created service account.
INFO Created cluster role.
INFO Created cluster role binding.
INFO Created Trident service.
INFO Created Trident statefulset.
INFO Created Trident daemonset.
INFO Waiting for Trident pod to start.
INFO Trident pod started.                          namespace=trident pod=trident-csi-0
INFO Waiting for Trident REST interface.
INFO Trident REST interface is up.                 version=19.01.0
INFO Trident installation succeeded.
```

It will look like this when the installer is complete:

```
# kubectl get pod -n trident
NAME                           READY     STATUS    RESTARTS   AGE
trident-csi-0                  4/4       Running   1          2m
trident-csi-vwhl2              2/2       Running   0          2m

# ./tridentctl -n trident version
+----------------+----------------+
| SERVER VERSION | CLIENT VERSION |
+----------------+----------------+
| 19.01.0        | 19.01.0        |
+----------------+----------------+
```

Using CSI Trident

To provision storage with CSI Trident, define one or more storage classes with the provisioner value `io.netapp.trident.csi`. The simplest storage class to start with is one based on the `sample-input/storage-class-csi.yaml.templ` file that comes with the installer, replacing `__BACKEND_TYPE__` with the storage driver name.:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: basic-csi
provisioner: io.netapp.trident.csi
parameters:
  backendType: "__BACKEND_TYPE__"
```

Note: `tridentctl` will detect and manage either Trident or CSI Trident automatically. We don't recommend installing both on the same cluster, but if both are present, use the `--csi` switch to force `tridentctl` to manage CSI Trident.

Uninstalling CSI Trident

Use the `--csi` switch to uninstall CSI Trident:

```
#./tridentctl uninstall --csi
INFO Deleted Trident daemonset.
INFO Deleted Trident statefulset.
INFO Deleted Trident service.
INFO Deleted cluster role binding.
INFO Deleted cluster role.
INFO Deleted service account.
INFO The uninstaller did not delete the Trident's namespace, PVC, and PV in case they␣
↪are going to be reused. Please use the --all option if you need the PVC and PV␣
↪deleted.
INFO Trident uninstallation succeeded.
```

CHAPTER 3

Design and Architecture Guide

3.1 Introduction

Containers have quickly become one of the most popular methods of packaging and deploying applications. The ecosystem surrounding the creation, deployment, and management of containerized applications has exploded, resulting in myriad solutions available to customers who simply want to deploy their applications with as little friction as possible.

Application teams love containers due to their ability to decouple the application from the underlying operating system. The ability to create a container on their own laptop, then deploy to a teammate's laptop, their on-premises data center, hyperscalars, and anywhere else means that they can focus their efforts on the application and its code, not on how the underlying operating system and infrastructure are configured.

At the same time, operations teams are only just now seeing the dramatic rise in popularity of containers. Containers are often approached first by developers for personal productivity purposes, which means the infrastructure teams are insulated from or unaware of their use. However, this is changing. Operations teams are now expected to deploy, maintain, and support infrastructures which host containerized applications. In addition, the rise of DevOps is pushing operations teams to understand not just the application, but the deployment method and platform at a much greater depth than ever before.

Fortunately there are robust platforms for hosting containerized applications. Arguably the most popular of those platforms is Kubernetes, an open source Cloud Native Computing Foundation (CNCF) project, which orchestrates the deployment of containers, including connecting them to network and storage resources as needed.

Deploying an application using containers doesn't change its fundamental resource requirements. Reading, writing, accessing, and storing data doesn't change just because a container technology is now a part of the stack in addition to virtual and/or physical machines.

To facilitate the consumption of storage resources by containerized applications, NetApp created and released an open source project known as Trident. Trident is a storage orchestrator which integrates with Docker and Kubernetes, as well as platforms built on those technologies, such as Red Hat OpenShift, Rancher, and IBM Cloud Private. The goal of Trident is to make the provisioning, connection, and consumption of storage as transparent and frictionless for applications as possible; while operating within the constraints put forth by the storage administrator.

To achieve this goal, Trident automates the storage management tasks needed to consume storage for the storage administrator, the Kubernetes and Docker administrators, and the application consumers. Trident fills a critical role for storage administrators, who may be feeling pressure from application teams to provide storage resources in ways which have not previously been expected. Modern applications, and just as importantly modern development practices, have changed the storage consumption model, where resources are created, consumed, and destroyed quickly. According to DataDog, containers have a median lifespan of just six days. This is dramatically different than storage resources for traditional applications, which commonly exist for years. Those which are deployed using container orchestrators have an even shorter lifespan of just a half day. Trident is the tool which storage administrators can rely on to safely, within the bounds given to it, provision the storage resources applications need, when they need them, and where they need them.

3.1.1 Target Audience

This document outlines the design and architecture considerations that should be evaluated when deploying containerized applications with persistence requirements within your organization. Additionally, you can find best practices for configuring Kubernetes and OpenShift with Trident.

It is assumed that you, the reader, have a basic understanding of containers, Kubernetes, and storage prior to reading this document. We will, however, explore and explain some of the concepts which are important to integrating Trident, and through it NetApp's storage platforms and services, with Kubernetes. Unless noted, Kubernetes and OpenShift can be used interchangeably in this document.

As with all best practices, these are suggestions based on the experience and knowledge of the NetApp team. Each should be considered according to your environment and targeted applications.

3.2 Concepts and Definitions

Kubernetes introduces several new concepts that storage, application and platform administrators should take into consideration. It is essential to understand the capability of each within the context of their use case.

3.2.1 Kubernetes storage concepts

The Kubernetes storage paradigm includes several entities which are important to each stage of requesting, consuming, and managing storage for containerized applications. At a high level, Kubernetes uses three types of objects to describe storage, as described below:

Persistent Volume claim

Persistent volume claims (PVCs) are used by applications to request access to storage resources. At a minimum, this includes two key characteristics:

- Size – The capacity desired by the application component
- Access mode – This describes the rules for accessing the storage volume. In particular, there are three access modes:
  - Read Write Once (RWO) – only one node is allowed to access the storage volume at a time for read and write access
  - Read Only Many (ROX) – many nodes may access the storage volume in read-only mode
  - Read Write Many (RWX) – many nodes may simultaneously read and write to the storage volume

- Optional: Storage Class - Which storage class to request for this request. See below for storage class information.

More information about PVCs can be found in the Kubernetes or OpenShift documentation.

Persistent Volume

PVs are objects that describe, to Kubernetes, how to connect to a storage device. Kubernetes supports many different types of storage. However, this document covers only NFS and iSCSI devices since NetApp platforms and services support those protocols. At a minimum, the PV must contain these parameters:

- The capacity represented by the object, e.g. "5Gi"
- The access mode–same as for PVCs–however, access modes can be dependent on protocol.
  - RWO is supported by all PVs
  - ROX is supported primarily by file and file-like protocols, e.g. NFS and CephFS. However, some block protocols are supported, such as iSCSI
  - RWX is supported by file and file-like protocols only, such as NFS
- The protocol, e.g. "iscsi" or "nfs", and additional information needed to access the storage. For example, an NFS PV will need the NFS server and mount path.
- A reclaim policy that describes the Kubernetes action when the PV is released. There are three options available:
  - Retain, which will mark the volume as waiting for administrator action. The volume cannot be reissued to another PVC.
  - Recycle, where, after being released, Kubernetes will connect the volume to a temporary pod and issue a `rm -rf` command to clear the data. For our interests, this is only supported by NFS volumes.
  - A policy of Delete will cause Kubernetes to delete the PV when it is released. Kubernetes does not, however, delete the storage which was referenced by the PV.

More information about PVs can be found in the Kubernetes or OpenShift documentation.

Storage Class

Kubernetes uses the storage class object to describe storage with specific characteristics. An administrator may define several storage classes that each define different storage properties. They are used by the *PVC* to provision storage. A storage class may have a provisioner associated with it that will trigger Kubernetes to wait for the volume to be provisioned by the specified provider. In the case of NetApp, the provisioner identifier used is `netapp.io/trident`.

A storage class object in Kubernetes has only two required fields:

- A name
- The provisioner, a full list of provisioners can be found in the documentation

The provisioner used may require additional attributes, which will be specific to the provisioner used. Additionally, the storage class may have a reclaim policy and mount options specified which will be applied to all volumes created for that storage class.

More information about storage classes can be found in the Kubernetes or OpenShift documentation.

3.2.2 Kubernetes Compute Concepts

In addition to basic storage concepts, like how to request and consume storage as described above, it's important to understand the compute concepts involved in the consumption of storage resources. Kubernetes is a container orchestrator, which means that it will dynamically assign containerized workloads to cluster members according to the resource requirements they have expressed (or defaults, if no explicit request is made).

For more information about what containers are and why they are different, see the Docker documentation.

Pods

A pod represents one or more containers which are related to each other. Containers which are members of the same pod are co-scheduled to the same node in the cluster. They typically share network and storage resources, though not every container in the pod may access the storage or be publicly accessible via the network.

The smallest granularity of management for Kubernetes compute resources is the pod. It is the atomic unit (smallest unit) of scale and is the consumer of other resources, such as storage.

Services

A Kubernetes service acts as an internal load balancer for replicated pods. It enables the scaling of pods while maintaining a consistent service IP address. There are several types of services, which may be reachable only within the cluster with a ClusterIP, or may be exposed to the outside world with a NodePort, LoadBalancer, or ExternalName.

Deployments

A deployment is one or more pods which are related to each other and often represent a "service" to a larger application being deployed. The application administrator uses deployments to declare the state of their application component and request that Kubernetes ensure that the state is implemented at all times. This can include several options:

- Pods which should be deployed, including versions, storage, network, and other resource requests
- Number of replicas of each pod instance

The application administrator then uses the deployment as the interface for managing the application. For example, by increasing or decreasing the number of replicas desired the application can be horizontally scaled in or out. Updating the deployment with a new version of the application pod(s) will trigger Kubernetes to remove existing instances one at a time and redeploy using the new version. Conversely, rolling back to a previous version of the deployment will cause Kubernetes to revert the pods to the previously specified version and configuration.

StatefulSets

Deployments specify how to scale application components, but it's limited to just the pods. When a webserver (which is managed as a Kubernetes deployment) is scaled up, Kubernetes will add more instances of that pod to reach the desired count. It is possible to add PVCs to deployments but then the PVC is shared by all pod replicas. What if each pod needs unique persistent storage?

StatefulSets are a special type of deployment where persistent storage is requested along with each replica of the pod(s). The StatefulSet definition includes a template PVC, which is used to request additional storage resources as the application is scaled out. In this case, each replica receives its own volume of storage. This is generally used for stateful applications such as databases.

In order to accomplish the above, StatefulSets provide unique pod names and network identifiers that are persistent across pod restarts. They also allow ordered operations, including startup, scale-up, upgrades, and deletion.

As the number of pod replicas increase, the number of PVCs do also. However, scaling down the application will not result in the PVCs being destroyed, as Kubernetes relies on the application administrator to clean up the PVCs in order to prevent inadvertent data loss.

3.2.3 Connecting containers to storage

When the application submits a PVC requesting storage, the Kubernetes engine will assign a PV which matches, or closely matches, the requirement. If no PV exists which can meet the request expressed in the PVC, then it will wait until a PV has been created which matches the request before making the assignment. If no storage class was assigned, then the Kubernetes administrator would be expected to request a storage resource and introduce a PV. However, the provisioner handles that process automatically when using storage classes.

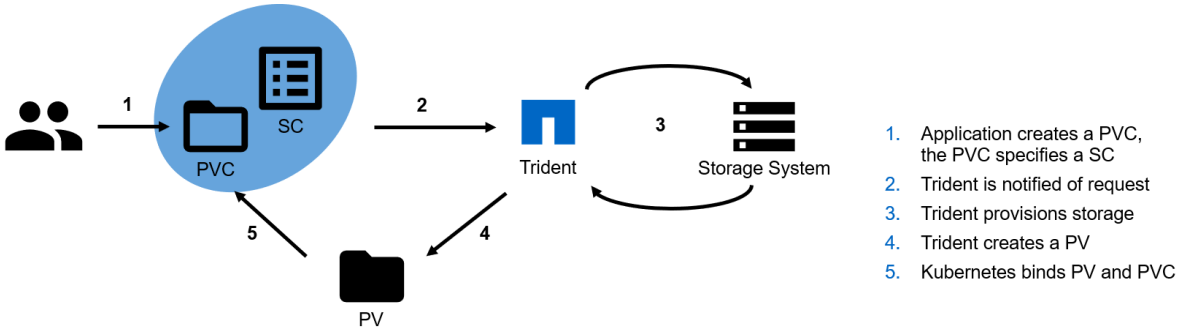

Fig. 1: Kubernetes dynamic storage provisioning process

The storage is not connected to a Kubernetes node within a cluster until the pod has been scheduled. At that time, `kubelet`, the agent running on each node that is responsible for managing container instances, mounts the storage to the host according to the information in the PV. When the container(s) in the pod are instantiated on the host, `kubelet` mounts the storage devices into the container.

3.2.4 Destroying and creating pods

It's important to understand that Kubernetes destroys and creates pods (workloads), it does not "move" them the same as live VM migration used by hypervisors. When Kubernetes scales down or needs to re-deploy a workload on a different host, the pod and the container(s) on the original host are stopped, destroyed, and the resources unmounted. The standard mount and instantiate process is then followed wherever in the cluster the same workload is re-deployed as a different pod with a different name, IP address, etc. When the application being deployed relies on persistent storage, that storage must be accessible from any Kubernetes node deploying the workload within the cluster. Without a shared storage system available for persistence, the data would be abandoned, and usually deleted, on the source system when the workload is re-deployed elsewhere in the cluster.

To maintain a persistent pod that will always be deployed on the same node with the same name and characteristics, a Stateful Set must be used as described above.

3.2.5 Container Storage Interface

The Cloud Native Computing Foundation (CNCF) is actively working on a standardized Container Storage Interface (CSI). NetApp is active in the CSI Special Interest Group (SIG). The CSI is meant to be a standard mechanism used by various container orchestrators to expose storage systems to containers. Trident v19.01 with CSI is currently in alpha

stage and runs with Kubernetes version <= 1.12. However, today CSI is in very early stages and does not provide the features NetApp's interface provides for Kubernetes. Therefore, NetApp recommends deploying Trident without CSI at this time and waiting until CSI is more mature.

3.3 NetApp Products and Integrations with Kubernetes

The NetApp portfolio of storage products integrates with many different aspects of a Kubernetes cluster, providing advanced storage capabilities which enhance the functionality, capability, performance, and availability of the Kubernetes deployment.

3.3.1 Trident

NetApp Trident is a dynamic storage provisioner for the containers ecosystem. It provides the ability to create storage volumes for containerized applications managed by Docker and Kubernetes. Trident is a fully supported, open source project hosted on GitHub. Trident works with the portfolio of NetApp storage platforms to deliver storage on-demand to applications according to policies defined by the administrator. When used with Kubernetes, Trident is deployed using native paradigms and provides persistent storage to all namespaces in the cluster. For more information about Trident, visit ThePub.

3.3.2 ONTAP

ONTAP is NetApp's multiprotocol, unified storage operating system providing advanced data management capabilities for any application. ONTAP systems may have all-flash, hybrid, or all-HDD configurations and offer many different deployment models, including engineered hardware (FAS and AFF), white-box (ONTAP Select), and cloud-only (Cloud Volumes ONTAP). Trident supports all the above mentioned deployment models.

3.3.3 Element OS

Element OS enables the storage administrator to consolidate workloads by guaranteeing performance and enabling a simplified and streamlined storage footprint. Coupled with a full API to enable automation of all aspects of storage management, Element OS enables storage administrators to do more with less effort.

Trident supports all Element OS clusters, more information can be found at Element Software.

3.3.4 NetApp HCI

NetApp HCI simplifies the management and scale of the datacenter by automating routine tasks and enabling infrastructure administrators to focus on more important functions.

NetApp HCI is fully supported by Trident, which is able to provision and manage storage devices for containerized applications directly against the underlying HCI storage platform. For more information about NetApp HCI visit NetApp HCI.

3.3.5 SANtricity

The NetApp E and EF Series storage platforms, using the SANtricity operating system, provides robust storage that is highly available, performant, and capable of delivering storage services for applications at any scale.

Trident is able to create and manage SANtricity volumes across the portfolio of products.

For more information about SANtricity and the storage platforms which use it, see SANtricity Software.

3.4 Kubernetes Cluster Architecture and Considerations

Kubernetes is extremely flexible and is capable of being deployed in many different configurations. It supports clusters as small as a single node and as large as a few thousand. It can be deployed using either physical or virtual machines on premises or in the cloud. However, single node deployments are mainly used for testing and are not suitable for production workloads. Also, hyperscalers such as AWS, Google Cloud and Azure abstract some of the initial and basic deployment tasks away. When deploying Kubernetes, there are a number of considerations and decisions to make which can affect the applications and how they consume storage resources.

3.4.1 Cluster concepts and components

A Kubernetes cluster typically consists of two types of nodes, each responsible for different aspects of functionality:

- Master nodes – These nodes host the control plane aspects of the cluster and are responsible for, among other things, the API endpoint which the users interact with and provide scheduling for pods across resources. Typically, these nodes are not used to schedule application workloads.
- Compute nodes – Nodes which are responsible for executing workloads for the cluster users.

The cluster has a number of Kubernetes intrinsic services which are deployed in the cluster. Depending on the service type, each service is deployed on only one type of node (master or compute) or on a mixture of node types. Some of these services, such as etcd and DNS, are mandatory for the cluster to be functional, while other services are optional. All of these services are deployed as pods within Kubernetes.

- etcd – etcd is a distributed key-value datastore. It is used heavily by Kubernetes to track the state and manage the resources associated with the cluster.
- DNS – Kubernetes maintains an internal DNS service to provide local resolution for the applications which have been deployed. This enables inter-pod communication to happen while referencing friendly names instead of internal IP addresses which can change as the container instances are scheduled.
- API Server - Kubernetes deploys the API server to allow interaction between kubernetes and the outside world. This is deployed on the master node(s).
- The dashboard – an optional component which provides a graphical interface to the cluster.
- Monitoring and logging – optional components which can aid with resource reporting.

Note: We have not discussed Kubernetes container networking to allow pods to communicate with each other, or to outside the cluster. The choice of using a overlay network (e.g. Flannel) or a straight layer 3 solution (e.g. Calico) is out of scope of this document and does not affect access to storage resources by the pods.

3.4.2 Cluster architectures

There are three primary Kubernetes cluster architectures. These accommodate various methods of high availablility and recoverability of the cluster, it's services, and the applications running. Trident is installed the same no matter which kubernetes architecture is chosen.

Master nodes are critical to the operation of the cluster. If no masters are running, or the master nodes are unable to reach a quorum, then the cluster is unable to schedule and execute applications. The master nodes are the control plane for the cluster and consequentially there should be special consideration given to their sizing and quantity.

Compute nodes are, generally speaking, much more disposable. However, extra resources must be built into the compute infrastructure to accomodate any workloads from failed nodes. Compute nodes can be added and removed from the cluster as needed quickly and easily to accommodate the scale of the applications which are being hosted. This makes it very easy to burst, and reclaim, resources based on real-time application workload.

Single master, compute

This architecture is the easiest to deploy but does not provide high availability of the core management services. In the event the master node is unavailable, no interaction can happen with the cluster until, at a minimum, the Kubernetes API server is returned to service.

This architecture can be useful for testing, qualification, proof-of-concept, and other non-production uses, however it should never be used for production deployments.

A single node used to host both the master service and the workloads is a variant of this architecture. Using a single node kubernetes cluster is useful when testing or experimenting with different concepts and capabilities. However, the limited scale and capacity make it unreasonable for more than very small tests. The Trident :ref: *quick start guide <Simple Kubernetes install>* outlines the process to instantiate a single node Kubernetes cluster with Trident that provides full functionality for testing and validation.

Multiple master, compute

Having multiple master nodes ensures that services remain available should master node(s) fail. In order to facilitate availability of master services, they should be deployed with odd numbers (e.g. 3,5,7,9 etc.) so quorum (master node majority) can be maintained should one or more masters fail. In the HA scenario, Kubernetes will maintain a copy of the etcd databases on each master, but hold elections for the control plane function leaders *kube-controller-manager* and *kube-scheduler* to avoid conflicts. The worker nodes can communicate with any master's API server through a load balancer.

> Deploying with multiple masters is the minimum recommended configuration for most production clusters.

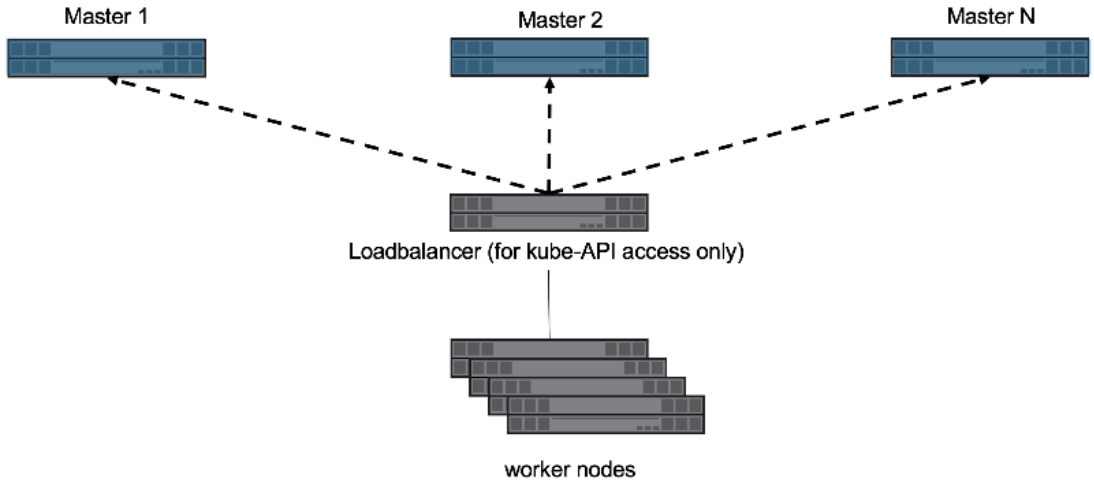

Fig. 2: Multiple master architecture

Pros:

- Provides highly available master services, ensuring that the loss of up to (n/2) – 1 master nodes will not affect cluster operations.

Cons:

- More complex initial setup.

Master, etcd, compute

This architecture isolates the etcd cluster from the other master server services. This removes workload from the master servers, enabling them to be sized smaller, and makes their scale out (or in) more simple. Deploying a Kubernetes cluster using this model adds a degree of complexity, however it adds flexibility to the scale, support, and management of the etcd service used by Kubernetes, which may be desirable to some organizations.

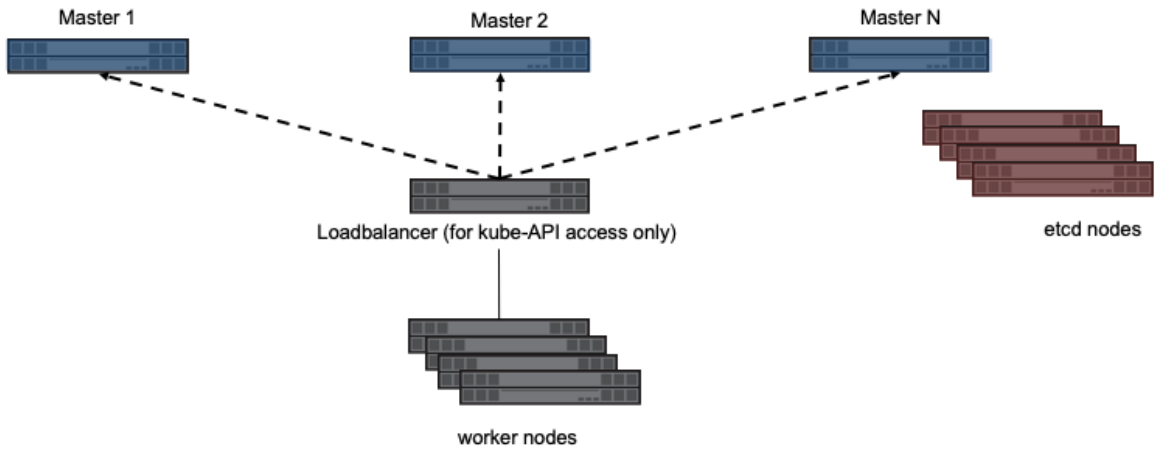

Fig. 3: Multiple master, etcd, compute architecture

Pros:

- Provides highly available master services, ensuring that the loss of up to (n/2) – 1 master nodes will not affect cluster operations.
- Isolating etcd from the other master services reduces workload for master servers.
- Decoupling etcd from the masters makes etcd administration and protection easier. Independent management allows for different protection and scaling schemes.

Cons:

- More complex initial setup.

Red Hat OpenShift infrastructure architecture

In addition to the architectures referenced above, Red Hat's OpenShift introduces the concept of infrastructure nodes. These nodes host cluster services such as log aggregation, metrics collection and reporting, container registry services, and overlay network management and routing.

Red Hat recommends a minimum of three infrastructure nodes for production deployments. This ensures that the services have resources available and are able to migrate in the event of host maintenance or failure.

This architecture enables the services which are critical to the cluster, i.e. registry, overlay network routing, and others to be hosted on dedicated nodes. These dedicated nodes may have additional redundancy, different CPU/RAM requirements, and other low-level differences from compute nodes. This also makes adding and removing compute nodes as needed easier, without needing to worry about core services being affected by a node being evacuated.

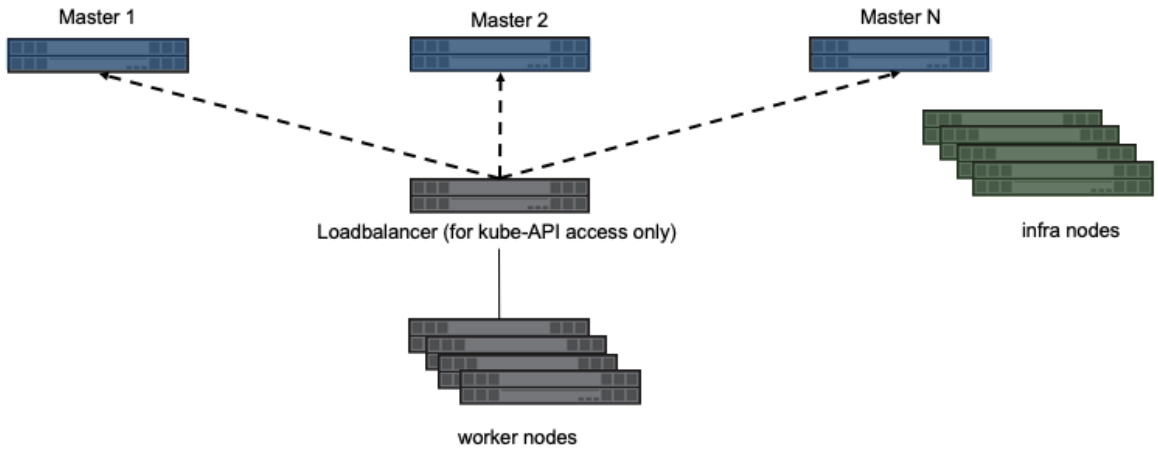

Fig. 4: OpenShift, Multiple master, infra, compute architecture

An additional option involves separating out the master and etcd roles into different servers in the same way as can be done in Kubernetes. This results in having master, etcd, infrastructure, and compute node roles. Further details, including examples of OpenShift node roles and potential deployment options, can be found in the Red Hat documentation.

3.4.3 Choosing an architecture

Regardless of the architecture that you choose, it's important to understand the ramifications to high availability, scalability, and serviceability of the component services. Be sure to consider the affect on the applications being hosted by the Kubernetes or OpenShift cluster. The architecture of the storage infrastructure supporting the Kubernetes/OpenShift cluster and the hosted applications can also be affected by the chosen cluster architecture, such as where etcd is hosted.

3.4.4 Persistent storage for cluster services

Dynamically provisioned persistent storage for the applications is provided using the storage class mechanism, with Trident acting as the interface to the NetApp portfolio. However, as you may have noted above there are several critical services hosted on the master servers and other cluster critical services which may be hosted on other node types.

Etcd persistent storage

When Kubernetes etcd is hosted by the master server it uses local storage. Instead, if you desire to leverage an enterprise storage array for etcd, you must mount a volume to the master node at the correct location prior to kubernetes

deployment. This storage cannot be dynamically provisioned by Trident or any other storage class provisioner as it is needed prior to the Kubernetes cluster being operational. This same paradigm holds true if dedicated etcd nodes are being used. Prior to deploying etcd, the volume from the storage system must be mounted to the host's file system at the location etcd is configured to use. Refer to your Kubernetes' provider documentation on where to mount the volume and/or customize the etcd configuration to use non-default storage.

Persistent storage for logging services

Centralized logging for applications deployed to the Kubernetes cluster is an optional service. Depending on how, and when, the service is deployed the storage class concepts may be able to dynamically provision storage for the service. Refer to your vendor's documentation on how to customize the storage for logging services. Additionally, this document discusses Red Hat's OpenShift logging service best practices in a later chapter.

Metrics and analytics services

Many third-party metrics and analytics tools are available for monitoring, reporting, and providing status of the applications and cluster. These tools may require persistent storage, often with specific performance characteristics. Each vendor has different storage recommendations and deployment methodology. Work with your vendor to identify requirements and, if needed, provision storage from an enterprise array to meet the requirements. This document will discuss the Red Hat OpenShift metrics service in a later chapter.

3.5 Storage for Kubernetes Infrastructure Services

Trident focuses on providing persistence to Kubernetes applications, but before you can host those applications, you need to run a healthy, protected Kubernetes cluster. Those clusters are made up of a number of services with their own persistence requirements that need to be considered.

Node-local container storage, a.k.a. graph driver storage

One of the often overlooked components of a Kubernetes deployment is the storage which the container instances consume on the Kubernetes cluster nodes, usually referred to as graph driver storage. When a container is instantiated on a node it consumes capacity and IOPS to do many of it's operations as only data which is read from or written to a persistent volume is offloaded to the external storage. If the Kubernetes nodes are expected to host dozens, hundreds, or more containers this may be a significant amount of temporary capacity and IOPS which are expected of the node-local storage.

Even if you don't have a requirement to keep the data, the containers still need enough performance and capacity to execute their application. The Kubernetes administrator and storage administrator should work closely together to determine the requirements for graph storage and ensure adequate performance and capacity is available.

The typical method of augmenting the graph driver storage is to use a block device mounted at the location where container instance storage is located, e.g. `/var/lib/docker`. The host operating system being used to underpin the Kubernetes deployment will each have different methods for how to replace the graph storage with something more robust than a simple directory on the node. Refer to the documentation from Red Hat, Ubuntu, SUSE, etc. for those instructions.

Note: Block protocol is specifically recommended for graph storage due to the nature of how the graph drivers work. In particular, they create thin clones, using a variety of methods depending on the driver, of the container image. NFS does not support this functionality and results in a full copy of the container image file system for each instance, resulting in significant performance and capacity implications.

If the Kubernetes nodes are virtualized, this could also be addressed by ensuring that the datastore they reside on meets the performance and capacity needs, however the flexibility of having a separate device, even an additional virtual disk, should be carefully considered. Using a separate device gives the ability to independently control capacity, performance, and data protection to tailor the policies according to needs. Often the capacity and performance needed for graph storage can fluctuate dramatically, however data protection is not necessary.

3.5.1 Persistent storage for cluster services

There are several critical services hosted on the master servers and other cluster critical services which may be hosted on other node types.

Using capacity provisioned from enterprise storage adds several benefits for each service as delineated below:

- Performance – leveraging enterprise storage means being able to provide latency in-line with application needs and controlling performance using QoS policies. This can be used to limit performance for bullies or ensure performance for applications as needed.
- Resiliency – in the event of node failure, being able to quickly recover the data and present it to a replacement ensures that the application is minimally affected.
- Data protection – putting application data onto dedicated volumes allows the data to have its own snapshot, replication, and retention policies.
- Data security – ensuring that data is encrypted and protected all the way to the disk level is important for meeting many compliance requirements.
- Scale – using enterprise storage enables deploying fewer instances, with the instance count depending on compute resources, rather than having to increase the total count for resiliency and performance purposes. Data which is protected automatically, and provides adequate performance, means that horizontal scale out doesn't need to compensate for limited performance.

There are some best practices which apply across all of the cluster services, or any application, which should be addressed as well.

- Determining the amount of acceptable data loss, i.e. the Recovery Point Objective (RPO), is a critical part of deploying a production level system. Having an understanding of this will greatly simplify the decisions around other items described in this section.
- Cluster service volumes should have a snapshot policy which enables the rapid recovery of data according to your requirements, as defined by the RPO. This enables quick and easy restoration of a service to a point in time by reverting the volume or the ability to recover individual files if needed.
- Replication can provide both backup and disaster recovery capabilities, each service should be evaluated and have the recovery plan considered carefully. Using storage replication may provide rapid recovery with a higher RPO, or can provide a starting baseline which expedites restore operations without having to recover all data from other locations.

etcd considerations

You should carefully consider the high availability and data protection policy for the etcd instance used by the Kubernetes master(s). This service is arguably the most critical to the overall functionality, recoverability, and serviceability of the cluster as a whole, so it's imperative that its deployment meets your goals.

The most common method of providing high availability and resiliency to the etcd instance is to horizontally scale the application by having multiple instances across multiple nodes. A minimum of three nodes is recommended.

Kubernetes etcd will, by default, use local storage for its persistence. This holds true whether etcd is co-located with other master services or is hosted on dedicated nodes. To use enterprise storage for etcd a volume must be provisioned

and mounted to the node at the correct location prior to deployment. This storage cannot be dynamically provisioned by Trident or any other storage class provisioner as it is needed prior to the Kubernetes cluster being operational.

Refer to your Kubernetes provider's documentation on where to mount the volume and/or customize the etcd configuration to use non-default storage.

logging

Centralized logging for applications deployed to the Kubernetes cluster is an optional service. Using enterprise storage for logging has the same benefits as with etcd: performance, resiliency, protection, security, and scale.

Depending on how and when the service is deployed, the storage class may define how to dynamically provision storage for the service. Refer to your vendor's documentation on how to customize the storage for logging services. Additionally, this document discusses Red Hat's OpenShift logging service best practices in a later chapter.

metrics

There are many third-party metrics and analytics services available for monitoring and reporting of the status and health of every aspect of the cluster and application. Many of these require persistent storage, often with specific performance characteristics, for the service in order for it to function as intended.

Architecturally, many of these function similarly where an agent exists on each node which forwards data to a centralized collector to aggregate, analyze, and display the data. Similar to the logging service, using entprise storage allows the aggregation service to move across nodes in the cluster seamlessly and ensures the data is protected in the event of node failure.

Each vendor has different recommendations and deployment methodology. Work with your vendor to identify requirements and, if needed, provision storage from an enterprise array to meet the requirements. This document will discuss the Red Hat OpenShift metrics service in a later chapter.

registry

The registry is the service with which users and applications will have the most direct interaction. It can also have a dramatic affect on the perceived performance of the Kubernetes cluster as a whole, as slow image push and pull operations can result in lengthy times for tasks which directly affect the developer and application.

Fortunately, the registry is flexible with regard to storage protocol. Keep in mind different protocols have different implications.

- Object storage is the default recommendation and is the simplest to use for Kubernetes deployments which expect to have significant scale or where the images need to be accessed across geographic regions.
- NFS is a a good choice for many deployments as it allows a single repository for the container images while allowing many registry endpoints to front the capacity.
- Block protocols, such as iSCSI, can be used for registry storage, but they introduce a single point of failure. The block device can only be attached to a single registry node due the single-writer limitation of the supported filesystems.

Protecting the images stored in the registry will have different priorities for each organization and each application. Registry images are, generally, either cached from upstream registries or have images pushed to them during the application build process. The RTO is important to the desired protection scheme because it will affect the recovery process. If RTO is not an issue, then the applications may be able to simply rebuild the container images and push them into a new instance of the registry. If faster RTO is desired, then a replication policy should be used which adheres to the desired recovery goal.

3.5.2 Design choices and guidelines when using ONTAP

When using ONTAP as the backend storage for containerized applications, with storage dynamically provisioned by Trident, there are several design and implementation considerations which should be addressed prior to deployment.

Storage Virtual Machines

Storage virtual machines (SVMs) are used for administrative delegation within ONTAP. They give the storage administrator the ability to isolate a particular user, group, or application to only having access to resources which they have been specifically granted. When Trident accesses the storage system via an SVM, it is prevented from doing many system level management tasks, providing additional isolation of capabilities for storage provisioning and management tasks.

There are several different ways which SVMs can be leveraged with Trident. Each is explained below. It's important to understand that having multiple Trident deployments, i.e. multiple Kubernetes clusters, does not change the below statements. When an SVM is shared with multiple Trident instances they simply need distinct prefixes defined in the backend configuration files.

SVM shared with non Trident-managed workloads

This configuration uses a single, or small number of, SVMs to host all of the workloads on the cluster and results in the containerized applications being hosted by the same SVM as other, non-containerized, workloads. The shared SVM model is common in organizations where there exists multiple network segments which are isolated and adding additional IP addresses is difficult or impossible.

There is nothing inherently wrong with this configuration, however it is more challenging to apply policies which affect only the container workloads.

Dedicated SVM for Trident-managed workloads

Creating an SVM which is used solely by Trident for provisioning and deprovisioning volumes for containerized workloads is the default recommendation from NetApp. This enables the storage administrator to put controls in place to limit the amount of resources which Trident is able to consume.

As was noted above, having multiple Kubernetes clusters connect to and consume storage from the same SVM is acceptable, the only change to the Trident configuration should be to *provide a different prefix*.

When creating backends which connect to the same underlying SVM resources, but have differing features applied, e.g. snapshot policies, using different prefixes is recommended to aid the storage administrator with identifying volumes and ensuring that no confusion ensues as a result.

Multiple SVMs dedicated to Trident-managed workloads

You may consider using multiple SVMs with Trident for many different reasons, including isolating applications and resource domains, strict control over resources, and to facilitate multitenacy. It's also worth considering using at least two SVMs with any Kubernetes cluster to isolate persistent storage for cluster services from application storage.

When using multiple SVMs, with one dedicated to cluster services, the goal is to isolate and control the workload in a more flexible way. This is possible because the expectation is that the Kubernetes cluster services SVM will not have dynamic provisioning happening against it in the same manner that the application SVM will. Many of the persistent storage resources needed by the Kubernetes cluster must exist prior to Trident deployment and consequentially must be manually provisioned by the storage administrator.

Kubernetes cluster services

Even for cluster services persistent volumes created by Trident, there should be serious consideration given to using per-volume QoS policies, including QoS minimums when possible, and customizing the volume options for the application. Below are the default recommendations for the cluster services, however you should evaluate your needs and

adjust policies according to your data protection, performance, and availability requirements. Despite these recommendations, you will still want to evaluate and determine what works best for your Kubernetes cluster and applications.

etcd

- The default snapshot policy is often adequate for protecting against data corruption and loss, however snapshots are not a backup strategy. Some consideration should be given to increasing the frequency, and decreasing the retention period, for etcd volumes. For example, keeping 24 hourly snapshots or 48 snapshots taken every 30 minutes, but not retaining them for more than one or two days. Since any data loss for etcd can be problematic, having more frequent snapshots makes this scenario easier to recover from.
- If the disaster recovery plan is to recover the Kubernetes cluster as-is at the destination site, then these volumes should be replicated with SnapMirror or SnapVault.
- etcd does not have significant IOPS or throughput requirements, however latency can play a critical role in the responsiveness of the Kubernetes API server. Whenever possible the lowest latency storage available should be used.
- A QoS policy should be leveraged to provide a minimum amount of IOPS to the etcd volume(s). The minimum value will depend on the number of nodes and pods which are deployed to your Kubernetes cluster. Monitoring should be used to verify that the configured policy is adequate and adjusted over time as the Kubernetes cluster expands.
- The etcd volumes should have their export policy or iGroup limited to only the nodes which are hosting, or could potentially host, etcd instances.

logging

- Volumes which are providing storage capacity for aggregated logging services need to be protected, however an average RPO is adequate in many instances since logging data is often not critical to recovery. If your application has strict compliance requirements, this may be different however.
- Using the default snapshot policy is generally adequate. Optionally, depending on the needs of the administrators, reducing the snapshot policy to one which keeps as few as seven daily snapshots may be acceptable.
- Logging volumes should be replicated to protect the historical data for use by the application and by administrators, however recovery may be deprioritized for other services.
- Logging services have widely variable IOPS requirements and read/write patterns. It's important to consider the number of nodes, pods, and other objects in the cluster. Each of these will generate data which needs to be stored, indexed, analyzed, and presented, so a larger cluster may have substantially more data than expected.
- A QoS policy may be leveraged to provide both a minimum and maximum amount of throughput available. Note that the maximum may need to be adjusted as additional pods are deployed, so close monitoring of the performance should be used to verify that logging is not being adversely affected by storage performance.
- The volumes export policy or iGroup should be limited to nodes which host the logging service. This will depend on the particular solution used and the chosen configuration. For example OpenShift's logging service is deployed to the infrastructure nodes.

metrics

- Kubernetes autoscale feature relies on metrics to provide data for when scale operations need to occur. Also, metrics data often plays a critical role in show-back and charge-back operations, so ensure that you are working to address the needs of the entire business with the RPO policy. Ensure that your RPO and RTO meet the needs of these functions.
- As the number of cluster nodes and deployed pods increases, so too does the amount of data which is collected and retained by the metrics service. It's important to understand the performance and capacity recommendations provided by the vendor for your metrics service as they can vary dramatically, particularly depending on the amount of time for which the data is retained and the number of metrics which are being monitored.

- A QoS policy can be used to limit the amount of IOPS or throughput which the metrics services uses, however it is generally not necessary to use a minimum policy.
- It is recommended to limit the export policy or iGroup to the hosts which the metrics service is executed from. Note that it's important to understand the architecture of your metrics provider. Many have agents which run on all hosts in the cluster, however those will report metrics to a centralised repository for storage and reporting. Only that group of nodes needs access.

registry

- Using a snapshot policy for the registry data may be valuable for recovering from data corruption or other issues, however it is not necessary. A basic snapshot policy is recommended, however individual container images cannot be recovered (they are stored in a hashed manner), only a full volume revert can be used to recover data.
- The workload for the registry can vary widely, however the general rule is that push operations happen infrequently, while pull operations happen frequently. If a CI/CD pipeline process is used to build, test, and deploy the application(s) this may result in a predictable workload. Alternatively, and even with a CI/CD system in use, the workload can vary based on application scaling requirements, build requirements, and even Kubernetes node add/remove operations. Close monitoring of the workload should be implemented to adjust as necessary.
- A QoS policy may be implemented to ensure that application instances are still able to pull and deploy new container images regardless of other workloads on the storage system. In the event of a disaster recovery scenario, the registry may have a heavy read workload while applications are instantiated on the destination site. The configured QoS minimum policy will prevent other disaster recovery operations from slowing application deployment.

3.6 Storage Configuration for Trident

Each storage platform in NetApp's portfolio has unique capabilities that benefit applications, containerized or not. Trident works with each of the major platforms: ONTAP, Element, and E-Series. There is not one platform which is better suited for all applications and scenarios than another, however the needs of the application and the team administering the device should be taken into account when choosing a platform.

The storage administrator(s), Kubernetes administrator(s), and the application team(s) should work with their NetApp team to ensure that the most appropriate platform is selected.

Regardless of which NetApp storage platform you will be connecting to your Kubernetes environment, you should follow the baseline best practices for the host operating system with the protocol that you will be leveraging. Optionally, you may want to consider incorporating application best practices, when available, with backend, storage class, and PVC settings to optimize storage for specific applications.

Some of the best practices guides are listed below, however refer to the NetApp Library for the most current versions.

- ONTAP
  - NFS Best Practice and Implementation Guide
  - SAN Administration Guide (for iSCSI)
  - iSCSI Express Configuration for RHEL
- SolidFire
  - Configuring SolidFire for Linux
- E-Series
  - Installing and Configuring for Linux Express Guide

Some example application best practices guides:

- ONTAP Best Practice Guidelines for MySQL
- MySQL Best Practices for NetApp SolidFire
- NetApp SolidFire and Cassandra
- Oracle Best Practices on NetApp SolidFire
- PostgreSQL Best Practices on NetApp SolidFire

Not all applications will have specific guidelines, it's important to work with your NetApp team and to refer to the library for the most up-to-date recommendations and guides.

3.6.1 Best practices for configuring ONTAP

The following recommendations are guidelines for configuring ONTAP for containerized workloads which consume volumes dynamically provisioned by Trident. Each should be considered and evaluated for appropriateness in your environment.

Use SVM(s) which are dedicated to Trident

Storage Virtual Machines (SVMs) provide isolation and administrative separation between tenants on an ONTAP system. Dedicating an SVM to applications enables the delegation of privileges and enables applying best practices for limiting resource consumption.

There are several options available for the management of the SVM:

- Provide the cluster management interface in the backend configuration, along with appropriate credentials, and specify the SVM name
- Create a dedicated management interface for the SVM
- Share the management role with an NFS data interface

In each case, the interface should be in DNS, and the DNS name should be used when configuring Trident. This helps to facilicate some DR scenarios, for example, SVM-DR without the use of network identity retention.

There is no preference between having a dedicated or shared management LIF for the SVM, however you should ensure that your network security policies align with the approach you choose. Regardless, the management LIF should be accessible via DNS to facilitate maximum flexibility should SVM-DR be used in conjunction with Trident.

Limit the maximum volume count

ONTAP storage systems have a maximum volume count which varies based on the software version and hardware platform, see the Hardware Universe for your specific platform and ONTAP version to determine exact limits. When the volume count is exhausted, provisioning operations will fail not only for Trident, but for all storage requests.

Trident's `ontap-nas` and `ontap-san` drivers provision a FlexVolume for each Kubernetes persistent volume (PV) which is created, and the `ontap-nas-economy` driver will create approximately one FlexVolume for every 200 PVs. To prevent Trident from consuming all available volumes on the storage system, it is recommended that a limit be placed on the SVM. This can be done from the command line:

```
vserver modify -vserver <svm_name> -max-volumes <num_of_volumes>
```

The value for `max-volumes` will vary based on several criteria specific to your environment:

- The number of existing volumes in the ONTAP cluster
- The number of volumes you expect to provision outside of Trident for other applications

- The number of persistent volumes expected to be consumed by Kubernetes applications

The `max-volumes` value is total volumes provisioned across all nodes in the cluster, not to an individual node. As a result, some conditions may be encountered where a cluster node may have far more, or less, Trident provisioned volumes than another.

For example, a 2-node ONTAP cluster has the ability to host a maximum of 2000 FlexVolumes. Having the maximum volume count set to 1250 appears very reasonable. However, if only aggregates from one node are assigned to the SVM, or the aggregates assigned from one node are unable to be provisioned against (e.g. due to capacity), then the other node will be the target for all Trident provisioned volumes. This means that the volume limit may be reached for that node before the `max-volumes` value is reached, resulting in impacting both Trident and other volume operations using that node. Avoid this situation by ensuring that aggregates from each node in the cluster are assigned to the SVM used by Trident in equal numbers.

In addition to controlling the volume count at the storage array, leveraging Kubernetes capabilities should also be used as explained in the next chapter.

Limit the maximum size of volumes created by the Trident user

ONTAP can prevent a user from creating a volume above a maximum size, as defined by the administrator. This is implemented using the permissions system and should be applied to the user which Trident uses to authenticate, e.g. `vsadmin`.

```
security login role modify -vserver <svm_name> -role <username> -access all -
↪cmddirname "volume create" -query "-size <=50g"
```

The above example command will prevent the user from creating volume larger than 50GiB in size. The value should be modified to what is appropriate for your applications and the expected size of volumes desired.

Note: This does not apply when using the `ontap-nas-economy` driver. The economy driver will create the FlexVolume with a size equal to the first PVC provisioned to that FlexVolume. Subsequent PVCs provisioned to that FlexVolume will result in the volume being resized, which is not subject to the limitation described above.

In addition to controlling the volume size at the storage array, leveraging Kubernetes capabilities should also be used as explained in the next chapter.

Create and use an SVM QoS policy

Leveraging an ONTAP QoS policy, applied to the SVM, limits the number of IOPS consumable by the Trident provisioned volumes. This helps to prevent a bully or out-of-control container from affecting workloads outside of the Trident SVM.

Creating a QoS policy for the SVM can be done in a few steps. Refer to the documentation for your version of ONTAP for the most accurate information. The example below creates a QoS policy which limits the total IOPS available to the SVM to 5000.

```
# create the policy group for the SVM
qos policy-group create -policy-group <policy_name> -vserver <svm_name> -max-
↪throughput 5000iops

# assign the policy group to the SVM, note this will not work
# if volumes or files in the SVM have existing QoS policies
vserver modify -vserver <svm_name> -qos-policy-group <policy_name>
```

Additionally, if your version of ONTAP supports it, you may consider using a QoS minimum in order to guarantee an amount of throughput to containerized workloads. Adaptive QoS is not compatible with an SVM level policy.

The number of IOPS dedicated to the containerized workloads depends on many aspects. Among other things, these include:

- Other workloads using the storage array. If there are other workloads, not related to the Kubernetes deployment, utilizing the storage resources, then care should be taken to ensure that those workloads are not accidentally adversely impacted.
- Expected workloads running in containers. If workloads which have high IOPS requirements will be running in containers, then a low QoS policy will result in a bad experience.

It's important to remember that a QoS policy assigned at the SVM level will result in all volumes provisioned to the SVM sharing the same IOPS pool. If one, or a small number, of the containerized applications has a high IOPS requirement it could become a bully to the other containerized workloads. If this is the case, you may want to consider using external automation to assign per-volume QoS policies.

Limit storage resource access to Kubernetes cluster members

Limiting access to the NFS volumes and iSCSI LUNs created by Trident is a critical component of the security posture for your Kubernetes deployment. Doing so prevents hosts which are not a part of the Kubernetes cluster from accessing the volumes and potentially modifying data unexpectedly.

It's important to understand that namespaces are the logical boundary for resources in Kubernetes. The assumption is that resources in the same namespace are able to be shared, however, importantly, there is no cross-namespace capability. This means that even though PVs are global objects, when bound to a PVC they are only accessible by pods which are in that same namespace. It's critical to ensure that namespaces are used to provide separation when appropriate.

The primary concern for most organizations, with regard to data security in a Kubernetes context, is that a process in a container can access storage mounted to the host, but which is not intended for the container. Simply put, this is not possible. The underlying technology for containers, namespaces, are designed to prevent this type of compromise. However, there is one exception: privileged containers.

A privileged container is one that is run with substantially more host-level permissions than normal. These are not denied by default, so disabling the capability using pod security policies is very important for preventing this accidental exposure.

For volumes where access is desired from both Kubernetes and external hosts, the storage should be managed in a traditional manner, with the PV introduced by the administrator and not managed by Trident. This ensures that the storage volume is destroyed only when both the Kubernetes and external hosts have disconnected and are no longer using the volume. Additionally, a custom export policy can be applied which enables access from the Kubernetes cluster nodes and targeted servers outside of the Kubernetes cluster.

For deployments which have dedicated infrastructure nodes (e.g. OpenShift), or other nodes which are not schedulable for user applications, separate export policies should be used to further limit access to storage resources. This includes creating an export policy for services which are deployed to those infrastructure nodes, for example the OpenShift Metrics and Logging services, and standard applications which are deployed to non-infrastructure nodes.

Create export policy

Create appropriate export policies for the Storage Virtual Machines. Allow only Kubernetes nodes access to the NFS volumes.

Export policies contain one or more export rules that process each node access request. Use the `vserver export-policy create` ONTAP CLI to create the export policy. Add rules to the export policy using the

`vserver export-policy rule create` ONTAP CLI command. Performing the above commands enables you to restrict which nodes have access to data.

Disable `showmount` for the application SVM

The showmount feature enables an NFS client to query the SVM for a list of available NFS exports. A pod deployed to the Kubernetes cluster could issue the showmount -e command against the data LIF and receive a list of available mounts, including those which it does not have access to. While this isn't, by itself, dangerous or a security compromise, it does provide unnecessary information potentially aiding an unauthorized user with connecting to an NFS export.

Disabling showmount is an SVM level command:

```
vserver nfs modify -vserver <svm_name> -showmount disabled
```

Use NFSv4 for Trident's etcd when possible

NFSv3 locks are handled by Network Lock Manager (NLM), which is a sideband mechanism not using the NFS protocol. Therefore, during a failure scenario and a server hosting the Trident pod ungracefully leaves the network (either by a hard reboot or all access being abruptly severed), the NFS lock is held indefinitely. This results in Trident failure because etcd's volume cannot be mounted from another node.

NFSv4 has session management and locking built into the protocol and the locks are released automatically when the session times out. In a recovery situation, the trident pod will be redeployed on another node, mount, and come back up after the v4 locks are automatically released.

3.6.2 Best practices for configuring SolidFire

Solidfire Account

Create a SolidFire account. Each SolidFire account represents a unique volume owner and receives its own set of Challenge-Handshake Authentication Protocol (CHAP) credentials. You can access volumes assigned to an account either by using the account name and the relative CHAP credentials or through a volume access group. An account can have up to two-thousand volumes assigned to it, but a volume can belong to only one account.

SolidFire QoS

Use QoS policy if you would like to create and save a standardized quality of service setting that can be applied to many volumes.

Quality of Service parameters can be set on a per-volume basis. Performance for each volume can be assured by setting three configurable parameters that define the QoS: Min IOPS, Max IOPS, and Burst IOPS.

The following table shows the possible minimum, maximum, and Burst IOPS values for 4Kb block Size.

IOPS Parameter	Definition	Min value	Default Value	Max Value(4Kb)
Min IOPS	The guaranteed level of performance for a volume.	50	50	15000
Max IOPS	Performance will not exceed this limit.	50	15000	200,000
Burst IOPS	Maximum IOPS allowed in a short burst scenario.	50	15000	200,000

Note: Although the Max IOPS and Burst IOPS can be set as high as 200,000, real-world maximum performance of a volume is limited by cluster usage and per-node performance.

Block size and bandwidth have a direct influence on the number of IOPS. As block sizes increase, the system increases bandwidth to a level necessary to process the larger block sizes. As bandwidth increases the number of IOPS the system is able to attain decreases. For more information on QoS and performance, refer to the NetApp SolidFire Quality of Service (QoS) Guide.

SolidFire authentication

Element supports two methods for authentication: CHAP and Volume Access Groups (VAG). CHAP uses the CHAP protocol to authenticate the host to the backend. Volume Access groups controls access to the volumes it provisions. NetApp recommends using CHAP for authentication as it's simplier and has no scaling limits.

CHAP authentication (verification that the initiator is the intended volume user) is supported only with account-based access control. If you are using CHAP for authentication, 2 options are available: unidirectional CHAP and bidirectional CHAP. Unidirectional CHAP authenticates volume access by using the SolidFire account name and initiator secret. The bidirectional CHAP option provides the most secure way of authenticating the volume since the volume authenticates the host through the account name and the initiator secret, and then the host authenticates the volume through the account name and the target secret.

However, if CHAP is unable to be enabled and VAGs are required, create the access group and add the host initiators and volumes to the access group. Each IQN that you add to an access group can access each volume in the group with or without CHAP authentication. If the iSCSI initiator is configured to use CHAP authentication, account-based access control is used. If the iSCSI initiator is not configured to use CHAP authentication, then volume access group access control is used.

For more information on how to setup Volume Access Groups and CHAP authentication, please refer the NetApp HCI Installation and setup guide.

3.7 Deploying Trident

The guidelines in this section provide recommendations for Trident installation with various Kubernetes configurations and considerations. As with all the other recommendations in this guide, each of these suggestions should be carefully considered to determine if it's appropriate and will provide benefit to your deployment.

3.7.1 Supported Kubernetes cluster architectures

Trident is supported with the following Kubernetes architectures. In each of the Kubernetes architectures below, the installation steps remain relatively the same except for the ones which have an asterick.

Kubernetes Cluster Architectures	Supported	Normal Installation
Single master, compute	Yes	Yes
Multiple master, compute	Yes	Yes
Master, etcd, compute	Yes*	No
Master, infrastructure, compute	Yes	Yes

The cell marked with an asterik above has an external production etcd cluster and requires different installation steps for deploying Trident. The *Trident etcd documentation* discusses in detail how to freshly deploy Trident on an external etcd cluster. It also mentions how to migrate existing Trident deployment to an external etcd cluster as well.

3.7.2 Trident installation modes

Three ways to install Trident are discussed in this chapter.

Normal install mode

Normal installation involves running the `tridentctl install -n trident` command which deploys the Trident pod on the Kubernetes cluster. Trident installation is quite a straightforward process. For more information on installation and provisioning of volumes, refer to the *Deploying documentation*.

Offline install mode

In many organizations, production and development environments do not have access to public repositories for pulling and posting images as these environments are completely secured and restricted. Such environments only allow pulling images from trusted private repositories. In such scenarios, make sure that a private registry instance is available. Then trident and etcd images should be downloaded from a bastion host with internet access and pushed on to the private registry. To install Trident in offline mode, just issue the `tridentctl install -n trident` command with the `--etcd-image` and the `--trident-image` parameter with the appropriate image name and location. For more information on how to install Trident in offline mode, please examine the blog on Installing Trident for Kubernetes from a Private Registry.

Remote install mode

Trident can be installed on a Kubernetes cluster from a remote machine. To do a remote install, install the appropriate version of `kubectl` on the remote machine from where you would be running the `tridentctl install` command remotely. Copy the configuration files from the Kubernetes cluster and set the KUBECONFIG environment variable on the remote machine. Initiate a `kubectl get nodes` command to verify you can connect to the required Kubernetes cluster. Complete the Trident deployment from the remote machine using the normal installation steps.

3.7.3 Trident CSI installation

The Container Storage Interface (CSI) is a standardized API for container orchestrators to manage storage plugins. CSI still in early stages and has not yet integrated much functionality. However, using the early specs, NetApp developed a Trident CSI for Kubernetes alpha driver for the Kubernetes CSI Beta integration for testing purposes only. As of Trident 19.01, NetApp recommends not deploying CSI in production environments until CSI becomes more mature. More information regarding CSI can be found in the *Trident documentation*.

3.7.4 Recommendations for all deployments

Deploy Trident to a dedicated namespace

Namespaces provide administrative separation between different applications and are a barrier for resource sharing, for example a PVC from one namespace cannot be consumed from another. Trident provides PV resources to all namespaces in the Kubernetes cluster and consequently leverages a service account which has elevated privileges.

Additionally, access to the Trident pod may enable a user to access storage system credentials and other sensitive information. It is important to ensure that application users and management applications do not have the ability to access the Trident object definitions or the pods themselves.

Use quotas and range limits to control storage consumption

Kubernetes has two features which, when combined, provide a powerful mechanism for limiting the resource consumption by applications. The storage quota mechanism allows the administrator to implement global, and storage class specific, capacity and object count consumption limits on a per-namespace basis. Further, using a range limit

will ensure that the PVC requests must be within both a minimum and maximum value before the request is forwarded to the provisioner.

These values are defined on a per-namespace basis, which means that each namespace will need to have values defined which fall in line with their resource requirements. An example of how to leverage quotas can be found on netapp.io.

Use PVC protection to protect in-use resources

This feature is on by default if running Kubernetes 1.10 or greater. Follow this process only if running Kubernetes 1.09 or less.

Storage object in use protection, or simply PVC protection, is a Kubernetes feature which prevents the deletion of PVCs which are in use by a pod and PVs which are bound to a PVC. This is important as it prevents a volume from being destroyed while it's actively being used by an application, which can result in data loss.

Implementing PVC protection is slightly different depending on your host operating system and Kubernetes distribution. Generically, the following process can be followed when using CentOS with "vanilla" Kubernetes, however be sure to follow the documentation for your particular operating system and Kubernetes version.

```
# from each master server in the cluster, edit the api server config
vi /etc/kubernetes/manifests/kube-apiserver.yaml

# search for the line under the "kube-apiserver" command stanza which
# starts with "- --admission-control" and append the value
# "StorageObjectInUseProtection"

# alternatively, backup the file and use the following sed command
cp /etc/kubernetes/manifests/kube-apiserver.yaml \
   /etc/kubernetes/manifests/kube-apiserver.yaml.orig

sed -i '/admission-control/ s/$/,StorageObjectInUseProtection/' \
  /etc/kubernetes/manifests/kube-apiserver.yaml

# after editing, restart the kube-apiserver service
systemctl restart kube-apiserver.service
```

After adding the admission controller for storage object protection you can verify it's functioning by viewing the details of a PVC and verifying the presence of the finalizer.

```
[root@kubemaster ~]# kubectl describe pvc pvc-protection
Name:          pvc-protection
Namespace:     default
StorageClass:  performance
Status:        Bound
Volume:        default-pvc-protection-3bcc8
Labels:        <none>
Annotations:   pv.kubernetes.io/bind-completed=yes
               pv.kubernetes.io/bound-by-controller=yes
               volume.beta.kubernetes.io/storage-provisioner=netapp.io/trident
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      3972844748800m
Access Modes:  RWO
Events:
  Type    Reason                Age                From                                                ␣
↪Message
  ----    ------                ----               ----                                                 ------
↪-
```

(continues on next page)

(continued from previous page)

```
  Normal  ExternalProvisioning  23s (x2 over 23s)  persistentvolume-controller
↪waiting for a volume to be created, either by external provisioner "netapp.io/
↪trident" or manually created by system administrator
  Normal  ProvisioningSuccess   21s                 netapp.io/trident
↪Kubernetes frontend provisioned a volume and a PV for the PVC
```

3.7.5 Deploying Trident to OpenShift

OpenShift uses Kubernetes for the underlying container orchestrator. Consequently, the same recommendations will apply when using Trident with Kubernetes or OpenShift. However, there are some minor additions when using OpenShift which should be taken into consideration.

Deploy Trident to infrastructure nodes

Trident is a core service to the OpenShift cluster, provisioning and managing the volumes used across all projects. Consideration should be given to deploying Trident to the infrastructure nodes in order to provide the same level of care and concern.

To deploy Trident to the infrastructure nodes, the project for Trident must be created by an administrator using the *oc adm* command. This prevents the project from inheriting the default node selector, which forces the pod to execute on compute nodes.

```
# create the project which Trident will be deployed to using
# the non-default node selector
oc adm new-project <project_name> --node-selector="region=infra"

# deploy Trident using the project name
tridentctl install -n <project_name>
```

The result of the above command is that any pod deployed to the project will be scheduled to nodes which have the tag "`region=infra`". This also removes the default node selector used by other projects which schedules pods to nodes which have the label "`node-role.kubernetes.io/compute=true`".

3.8 Integrating Trident

3.8.1 Trident backend design

ONTAP

Choosing a backend driver for ONTAP

Four different backend drivers are available for ONTAP systems. These drivers are differentiated by the protocol being used and how the volumes are provisioned on the storage system. Therefore, give careful consideration regarding which driver to deploy.

At a higher level, if your application has components which need shared storage (multiple pods accessing the same PVC) NAS based drivers would be the default choice, while the block based iSCSI driver meets the needs of non-shared storage. Choose the protocol based on the requirements of the application and the comfort level of the storage and infrastructure teams. Generally speaking, there is little difference between them for most applications, so often the decision is based upon whether or not shared storage (where more than one pod will need simultaneous access) is needed.

The four *drivers* for ONTAP backends are listed below:

- `ontap-nas` – each PV provisioned is a full ONTAP FlexVolume
- `ontap-nas-economy` – each PV provisioned is a qtree, with up to 200 qtrees per FlexVolume
- `ontap-nas-flexgroup` - each PV provisioned as a full ONTAP FlexGroup, and all aggregates assigned to a SVM are used.
- `ontap-san` – each PV provisioned is a LUN within its own FlexVolume

Choosing between the three NFS drivers has some ramifications to the features which are made available to the application.

Note that, in the tables below, not all of the capabilities are exposed through Trident. Some must be applied by the storage administrator after provisioning if that functionality is desired. The superscript footnotes distinguish the functionality per feature and driver.

Table 1: ONTAP NAS driver capabilities

ONTAP NFS Drivers	Snapshots	Clones	Multi-attach	QoS	Resize	Replication
`ontap-nas`	Yes	Yes	Yes	Yes[2]	Yes	Yes[2]
`ontap-nas-economy`	Yes[1]	No	Yes	Yes[12]	Yes	Yes[2]
`ontap-nas-flexgroup`	Yes	No	Yes	Yes[2]	Yes	Yes[2]

The SAN driver capabilities are shown below.

Table 2: ONTAP SAN driver capabilities

ONTAP SAN Driver	Snapshots	Clones	Multi-attach	QoS	Resize	Replication
`ontap-san`	Yes	Yes	No	Yes[2]	Yes[2]	Yes[2]

Footnote for above tables:
Yes[1]: Trident managed, but not PV granular
Yes[2]: Not Trident managed
Yes[12]: Not Trident managed and not PV granular

The features that are not PV granular are applied to the entire FlexVolume and all of the PVs (i.e. qtrees) will share a common schedule for each qtree.

As we can see in the above tables, much of the functionality between the `ontap-nas` and `ontap-nas-economy` is the same. However, since the `ontap-nas-economy` driver limits the ability to control the schedule at per-PV granularity, this may affect your disaster recovery and backup planning in particular. For development teams which desire to leverage PVC clone functionality on ONTAP storage, this is only possible when using the `ontap-nas` or `ontap-san` drivers (note, the `solidfire-san` driver is also capable of cloning PVCs).

SolidFire Backend Driver

The `solidfire-san` driver, used with the SolidFire platform, helps the admin configure a SolidFire backend for Trident on the basis of QoS limits. If you would like to design your backend to set the specific QoS limits on the volumes provisioned by Trident, use the *Type* parameter in the backend file. The admin also can restrict the volume size that could be created on the storage using the *limitVolumeSize* parameter. Currently SolidFire storage features like volume resize and volume replication are not supported through the `solidfire-san` driver. These operation should be done manually through Element OS Web UI.

Table 3: SolidFire SAN driver capabilities

SolidFire Driver	Snapshots	Clones	Multi-attach	QoS	Resize	Replication
`solidfire-san`	Yes	Yes	No	Yes	Yes[1]	Yes[1]

Footnote:
Yes[1]: Not Trident managed

3.8.2 Storage Class design

Individual Storage Classes need to be configured and applied to create a Kubernetes storage class object. This section discusses how to design a storage class for your application.

Storage Class design For specific backend utilization

Filtering can be used within a specific storage class object to determine which storage pool or set of pools are to be used with that specific storage class. Three sets of filters can be set in the Storage Class: *storagePools*, *additionalStoragePools*, and/or *excludeStoragePools*.

The *storagePools* parameter helps restrict storage to the set of pools that match any specified attributes. The *additionalStoragePools* parameter is used to extend the set of pools that Trident will use for provisioning along with the set of pools selected by the attributes and *storagePools* parameters. You can use either parameter alone or both together to make sure that appropriate set of storage pools are selected.

The *excludeStoragePools* parameter is used to specifically exclude the listed set of pools that match the attributes.

Please refer to *Trident StorageClass Objects* on how these parameters are used.

Storage Class design To emulate QoS policies

If you would like to design Storage Classes to emulate Quality of Service policies, create a Storage Class with the *media* attribute as *hdd* or *ssd*. Based on the *media* attribute mentioned in the storage class, Trident will select the appropriate backend that serves *hdd* or *ssd* aggregates to match the media attribute and then direct the provisioning of the volumes on to the specific aggregate. Therefore we can create a storage class PREMIUM which would have *media* attribute set as *ssd* which could be classified as the PREMIUM QoS policy. We can create another storage class STANDARD which would would have the media attribute set as 'hdd' which could be classified as the STANDARD QoS policy. We could also use the "IOPS" attribute in the storage class to redirect provisioning to a SolidFire appliance which can be defined as a QoS Policy.

Please refer to *Trident StorageClass Objects* on how these parameters can be used.

Storage Class Design To utilize backend based on specific features

Storage Classes can be designed to direct volume provisioning on a specific backend where features such as thin and thick provisioning, snapshots, clones and encryption are enabled. To specify which storage to use, create Storage Classes that specify the appropriate backend with the required feature enabled.

Please refer to *Trident StorageClass Objects* on how these parameters can be used.

3.8.3 PVC characteristics which affect storage provisioning

Some parameters beyond the requested storage class may affect Trident's provisioning decision process when creating a PVC.

Access mode

When requesting storage via a PVC, one of the mandatory fields is the access mode. The mode desired may affect the backend selected to host the storage request.

Trident will attempt to match the storage protocol used with the access method specified according to the following matrix. This is independent of the underlying storage platform.

Table 4: Protocols used by access modes

	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
iSCSI	Yes	Yes	No
NFS	Yes	Yes	Yes

A request for a ReadWriteMany PVC submitted to a Trident deployment without an NFS backend configured will result in no volume being provisioned. For this reason, the requestor should use the access mode which is appropriate for their application.

3.8.4 Modifying persistent volumes

Persistent volumes are, with two exceptions, immutable objects in Kubernetes. Once created, the reclaim policy and the size can be modified. However, this doesn't prevent some aspects of the volume from being modified outside of Kubernetes. This may be desirable in order to customize the volume for specific applications, to ensure that capacity is not accidentally consumed, or simply to move the volume to a different storage controller for any reason.

Note: Kubernetes in-tree provisioners do not support volume resize operations for NFS or iSCSI PVs at this time. Trident supports expanding NFS volumes. For a list of PV types which support volume resizing refer to the Kubernetes documentation.

The connection details of the PV cannot be modified after creation.

Volume move operations

Storage administrators have the ability to move volumes between aggregates and controllers in the ONTAP cluster non-disruptively to the storage consumer. This operation does not affect Trident or the Kubernetes cluster, so long as the destination aggregate is one which the SVM Trident is using has access to. Importantly, if the aggregate has been newly added to the SVM, the backend will need to be "refreshed" by re-adding it to Trident. This will trigger Trident to reinventory the SVM so that the new aggregate is recognized.

However, moving volumes across backends is not supported. This includes between SVMs in the same cluster, between clusters, or onto a different storage platform (even if that storage system is one which is connected to Trident).

Resizing volumes

To make sure that the Persistent Volumes provisioned by Trident can be resized later, create Persistent Volume based out of a PersistentVolume Claim that utilizes a Storage Class which allow volume expansion by setting "allowVolumeExpansion" attribute as true. Whenever the Persistent Volume needs to be resized, edit the

"spec.resources.requests.storage" annotation in the Persistent Volume Claim to the required volume size and Trident will automatically take care of resizing the volume on ONTAP.

Note:

1. Currently NFS PV resize is only supported by Trident and not iSCSI PV resize.
2. Kubernetes, prior to version 1.12, does not support NFS PV resize as the admission controller may reject PVC size updates. The Trident team has changed Kubernetes to allow such changes starting with Kubernetes 1.12. While we recommend using Kubernetes 1.12, it is still possible to resize NFS PVs for earlier versions of Kubernetes that support resize. This is done by disabling the PersistentVolumeClaimResize admission plugin when the Kubernetes API server is started.

3.8.5 When to manually provision a volume instead of using Trident

Trident's goal is to be the provisioning engine for all storage consumed by containers. However, we understand that there are scenarios which may still need a manually provisioned PV. Generally speaking, these situations are limited to needing to customize the properties of the underlying storage device in ways which Trident does not support.

There are two ways which the desired settings can be applied:

1. Use the backend configuration, or PVC attributes, to customize the volume properties at provisioning time
2. After the volume is provisioned, the storage administrator applies configuration to the volume which is bound to the PVC

Option number 1 is limited by the volume options with Trident supports, which do not encompass all of the options available. Option 2 may be the only viable solution for fully customizing storage for a particular application. Finally, you can always provision a volume manually and introduce a matching PV outside of Trident if you do not want Trident to manage it for some reason.

If you have requirements to customize volumes in ways which Trident does not support, please let us know using resources on the contact_us page.

3.8.6 Deploying OpenShift services using Trident

The OpenShift value-add cluster services provide important functionality to cluster administrators and the applications being hosted. The storage which these services use can be provisioned using the node-local resources, however this often limits the capacity, performance, recoverability, and sustainability of the service. Leveraging an enterprise storage array to provide capacity to these services can enable dramatically improved service, however, as with all applications, the OpenShift and storage administrators should work closely together to determine the best options for each. The Red Hat documentation should be leveraged heavily to determine the requirements and ensure that sizing and performance needs are met.

Registry service

Deploying and managing storage for the registry has been documented on netapp.io in this blog post.

Logging service

Like other OpenShift services, the logging service is deployed using Ansible with configuration parameters supplied by the inventory, a.k.a. hosts, file provided to the playbook. There are two installation methods which will be covered: deploying logging during initial OpenShift install and deploying logging after OpenShift has been installed.

Warning: As of Red Hat OpenShift version 3.9, the official documentation recommends against NFS for the logging service due to concerns around data corruption. This is based on Red Hat testing of their products. ONTAP's NFS server does not have these issues, and can easily back a logging deployment. Ultimately, the choice of protocol for the logging service is up to you, just know that both will work great when using NetApp platforms and there is no reason to avoid NFS if that is your preference.

If you choose to use NFS with the logging service, you will need to set the Ansible variable `openshift_enable_unsupported_configurations` to `true` to prevent the installer from failing.

Getting started

The logging service can, optionally, be deployed for both applications as well as for the core operations of the OpenShift cluster itself. If you choose to deploy operations logging, by specifying the variable `openshift_logging_use_ops` as `true`, two instances of the service will be created. The variables which control the logging instance for operations contain "ops" in them, whereas the instance for applications do not.

Configuring the Ansible variables according to the deployment method is important in order to ensure that the correct storage is utilized by the underlying services. Let's look at the options for each of the deployment methods

Note: The tables below only contain the variables which are relevant for storage configuration as it relates to the logging service. There are many other options found in the documentation which should be reviewed, configured, and used according to your deployment.

The variables in the below table will result in the Ansible playbook creating a PV and PVC for the logging service using the details provided. This method is significantly less flexible than using the component installation playbook after OpenShift installation, however if you have existing volumes available, it is an option.

Table 5: Logging variables when deploying at OpenShift install time

Variable	Details
`openshift_logging_storage_kind`	Set to `nfs` to have the installer create an NFS PV for the logging service.
`openshift_logging_storage_host`	The hostname or IP address of the NFS host. This should be set to the data LIF for your virtual machine.
`openshift_logging_storage_nfs_directory`	The mount path for the NFS export. For example, if the volume is junctioned as `/openshift_logging`, you would use that path for this variable.
`openshift_logging_storage_volume_name`	The name, e.g. `pv_ose_logs`, of the PV to create.
`openshift_logging_storage_volume_size`	The size of the NFS export, for example `100Gi`.

If your OpenShift cluster is already running, and therefore Trident has been deployed and configured, the installer can use dynamic provisioning to create the volumes. The following variables will need to be configured.

Table 6: Logging variables when deploying after OpenShift install

Variable	Details
`openshift_logging_es_pvc_dynamic`	Set to true to use dynamically provisioned volumes.
`openshift_logging_es_pvc_storage_class_name`	The name of the storage class which will be used in the PVC.
`openshift_logging_es_pvc_size`	The size of the volume requested in the PVC.
`openshift_logging_es_pvc_prefix`	A prefix for the PVCs used by the logging service.
`openshift_logging_es_ops_pvc_dynamic`	Set to `true` to use dynamically provisioned volumes for the ops logging instance.
`openshift_logging_es_ops_pvc_storage_class_name`	The name of the storage class for the ops logging instance.
`openshift_logging_es_ops_pvc_size`	The size of the volume request for the ops instance.
`openshift_logging_es_ops_pvc_prefix`	A prefix for the ops instance PVCs.

Note: A bug exists in OpenShift 3.9 which prevents a storage class from being used when the value for `openshift_logging_es_ops_pvc_dynamic` is set to `true`. However, this can be worked around by, counterintuitively, setting the variable to `false`, which will include the storage class in the PVC.

Deploy the logging stack

If you are deploying logging as a part of the initial OpenShift install process, then you only need to follow the standard deployment process. Ansible will configure and deploy the needed services and OpenShift objects so that the service is available as soon as Ansible completes.

However, if you are deploying after the initial installation, the component playbook will need to be used by Ansible. This process may change slightly with different versions of OpenShift, so be sure to read and follow the documentation for your version.

Metrics service

The metrics service provides valuable information to the administrator regarding the status, resource utilization, and availability of the OpenShift cluster. It is also necessary for pod autoscale functionality and many organizations use data from the metrics service for their charge back and/or show back applications.

Like with the logging service, and OpenShift as a whole, Ansible is used to deploy the metrics service. Also, like the logging service, the metrics service can be deployed during initial setup of the cluster or after its operational using the component installation method. The following tables contain the variables which are important when configuring persistent storage for the metrics service.

Note: The tables below only contain the variables which are relevant for storage configuration as it relates to the metrics service. There are many other options found in the documentation which should be reviewed, configured, and used according to your deployment.

Table 7: Metrics variables when deploying at OpenShift install time

Variable	Details
`openshift_metrics_storage_kind`	Set to `nfs` to have the installer create an NFS PV for the logging service.
`openshift_metrics_storage_host`	The hostname or IP address of the NFS host. This should be set to the data LIF for your SVM.
`openshift_metrics_storage_nfs_directory`	The mount path for the NFS export. For example, if the volume is junctioned as `/openshift_metrics`, you would use that path for this variable.
`openshift_metrics_storage_volume_name`	The name, e.g. `pv_openshift_metrics`, of the PV to create.
`openshift_metrics_storage_volume_size`	The size of the NFS export, for example `100Gi`.

If your OpenShift cluster is already running, and therefore Trident has been deployed and configured, the installer can use dynamic provisioning to create the volumes. The following variables will need to be configured.

Table 8: Metrics variables when deploying after OpenShift install

Variable	Details
`openshift_metrics_cassandra_pvc_prefix`	A prefix to use for the metrics PVCs.
`openshift_metrics_cassandra_pvc_size`	The size of the volumes to request.
`openshift_metrics_cassandra_storage_type`	The type of storage to use for metrics, this must be set to dynamic for Ansible to create PVCs with the appropriate storage class.
`openshift_metrics_cassanda_pvc_storage_class_name`	The name of the storage class to use.

Deploying the metrics service

With the appropriate Ansible variables defined in your hosts/inventory file, deploy the service using Ansible. If you are deploying at OpenShift install time, then the PV will be created and used automatically. If you're deploying using the component playbooks, after OpenShift install, then Ansible will create any PVCs which are needed and, after Trident has provisioned storage for them, deploy the service.

The variables above, and the process for deploying, may change with each version of OpenShift. Ensure you review and follow the deployment guide for your version so that it is configured for your environment.

3.9 Backup and Disaster Recovery

Protecting application data is one of the fundamental purposes of any storage system. Regardless of an application being cloud native, 12 factor, microservice or any other architecture, the data still needs to be protected for the application to continue to function and be valuable to the organization.

NetApp's storage platforms provide data protection and recoverability options which vary based on recovery time and acceptable data loss requirements. Trident can provision volumes which can take advantage of some of these features, however a full data protection and recovery strategy should be evaluated for each application with a persistence requirement.

3.9.1 ONTAP snapshots

Snapshots play an important role by providing point-in-time recovery options for application data. It's important to remember, however, that snapshots are not backups by themselves. They will not protect against storage system failure or other catastrophes which result in the storage system failing. Snapshots are a convenient, quick, and easy way to recover data for most scenarios.

Using ONTAP snapshots with containers

A backend which has not explicitly set a snapshot policy will use the "`none`" policy. This means that ONTAP will not take any snapshots of the volume automatically. If the storage administrator takes manual snapshots or changes the snapshot policy via the ONTAP management interface, this will not affect Trident operation.

Note that the snapshot directory is hidden by default. This is to facilitate maximum compatibility of volumes provisioned using the ontap-nas and ontap-nas-economy drivers as some applications, such as MySQL, could experience issues.

Accessing the snapshot directory

The `.snapshot` directory is a mechanism which can be enabled when using the `ontap-nas` and `ontap-nas-economy` drivers to allow applications to recover data from snapshots directly. More information about how to enable access and enable self-service data recovery can be found in this blog post on netapp.io.

Restoring the snapshots

It is possible to restore a volume to a state recorded in a previously created snapshot copy to retrieve lost information using the "volume snapshot restore" ONTAP CLI command . When you restore a Snapshot copy, the restore operation overwrites the existing volume configuration. Any changes made to the data in the volume after the Snapshot copy was created are lost.

```
cluster1::*> volume snapshot restore -vserver vs0 -volume vol3 -snapshot vol3_snap_
↪archive
```

3.9.2 SolidFire snapshots

It is possible to backup data on a SolidFire Volume by setting a snapshot schedule to a SolidFire volume. This would make sure that the snapshots of the volume are taken at the required interval. However, it is not possible to set a snapshot schedule to a volume through the solidfire-san driver. This would have to be set manually using the Element OS Web UI or Element OS APIs.

In the event of a data corruption, we can choose a particular snapshot and rollback the volume to the snapshot manually using the Element OS Web UI or Element OS APIs. This reverts any changes made to the volume since the snapshot was created.

3.9.3 Etcd snapshots using `etcdctl` command line utility

The etcdctl command line utility offers the provision to take snapshot of an etcd cluster. It also enables us to restore the previously taken snapshot.

etcdctl snapshot backup

The etcdctl command `etcdctl snapshot save /var/etcd/data/snapshot.db` enables us to take a point-in-time snapshot of the etcd cluster. NetApp recommends using a script to take timely backups. This command can be deployed from within the etcd container or the command can be deployed using the `kubectl exec` command directly. Store the periodic snapshots under the persistent Trident NetApp volume */var/etcd/data* so that snapshots are stored securely and can safely be recovered should the trident pod be lost. Periodically check the volume to be sure it does not run out of space.

etcdctl snapshot restore

In the event of an accidental deletion or corruption of Trident etcd data, we can choose the appropriate snapshot and restore it back using the command `etcdctl snapshot restore snapshot.db --data-dir /var/etcd/data/etcd-test2 --name etcd1`. Take note to restore the snapshot on to a different folder [shown in the above command as /var/etcd/data/etcd-test2 which is on the mount] inside the Trident NetApp volume.

After the restore is complete, uninstall Trident. Take note not to use the "-a" flag during uninstallation. Mount the Trident volume manually on the host and make sure that the current "member" folder under /var/etcd/data is deleted. Copy the "member" folder from the restored folder /var/etcd/data/etcd-test2 to /var/etcd/data. After the copy is complete, re-install Trident. Verify if the restore and recovery has been completed successfully by making sure all the required data is present.

3.9.4 Data replication using ONTAP

Replicating data can play an important role in protecting against data loss due to storage array failure. Snapshots are a point-in-time recovery which provide a very quick and easy method of recovering data which has been corrupted or accidentally lost as a result of human or technological error. However, they cannot protect against catastrophic failure of the storage array itself. Trident is unable to configure replication relationships itself, however the storage administrator can use ONTAP's SVM-DR function to automatically replicate volumes to a DR destination. If this method is used to automatically protect Trident provisioned volumes, there are some considerations to take into account.

- A distinct backend should be created for each SVM which has SVM-DR enabled.

- Storage classes should be crafted so as to not select the replicated backends except when desired. This is important to avoid having volumes which do not need the protection of a replication relationship be provisioned onto the backend.
- Application administrators should understand the additional cost and complexity associated with replicating the data and a plan for recovery should be determined before they leverage replication.
- Trident does not automatically detect SVM failures. Therefore, upon a failure, the administrator needs to run the command *tridentctl backend update* to trigger Trident's failover to the new backend.

3.10 Security Recommendations

3.10.1 Run Trident in its own namespace

It is important to prevent applications, application admins, users, and management applications from accessing Trident object definitions or the pods to ensure reliable storage and block potential malicious activity. To separate out the other applications and users from Trident, always install Trident in its own Kubernetes namespace. In our Installing Trident docs we call this namespace *trident*. Putting Trident in its own namespace assures that only the Kubernetes administrative personnel have access to the Trident pod and the artifacts (such as backend and CHAP secrets if applicable) stored in Trident's etcd datastore. Allow only administrators access to the trident namespace and thus access to *tridentctl* application.

3.10.2 etcd container

Trident's state is stored in etcd. Etcd contains details regarding the backends, storage classes, and volumes the Trident provisioner creates. Trident's default install method installs etcd as a container in the Trident pod. For security reasons, the etcd container located within the Trident pod is not accessible via its REST interface outside the pod, and only the trident container can access the etcd container. If you choose to use an *external etcd*, authenticate Trident with the etcd cluster using certificates. This also ensures all the communication between the Trident and etcd is encrypted.

3.10.3 CHAP authentication

NetApp recommends deploying bi-directional CHAP to ensure authentication between a host and the SolidFire backend. Trident uses a secret object that includes two CHAP passwords per SolidFire tenant. Kubernetes manages mapping of Kubernetes tenant to SF tenant. Upon volume creation time, Trident makes an API call to the SolidFire system to retrieve the secrets if the secret for that tenant doesn't already exist. Trident then passes the secrets on to Kubernetes. The kubelet located on each node accesses the secrets via the Kubernetes API and uses them to run/enable CHAP between each node accessing the volume and the SolidFire system where the volumes are located.

Since Trident v18.10, SolidFire defaults to use CHAP if the Kubernetes version is >= 1.7 and a Trident access group doesn't exist. Setting *AccessGroup* or *UseCHAP* in the backend configuration file overrides this behavior. CHAP is guaranteed by setting `UseCHAP` to `true` in the backend.json file.

3.10.4 Storage backend credentials

Delete backend config files

Deleting the backend config files helps prevent unauthorized users from accessing backend usernames, passwords and other credentials. After the backends are created using the *tridentctl create backend* command, make sure that the backend config files are deleted from the node where Trident was installed. Keep a copy of the file in a secure location if the backend file will be updated in the future (e.g. to change passwords)..

Encrypt Trident* etcd Volume

Since Trident stores all its backend credentials in its etcd datastore, it may be possible for unauthorized users to access this information either from etcdctl or directly from the hardware hosting the etcd volume. Therefore, as an additional security measure, enable encryption on the Trident volume.

The appropriate encryption license must be enabled on the backends to encrypt Tridents volume.

ONTAP Backend

> Prior to Trident installation, edit the temporary backend.json file to include *encryption*. When the volume is created for Tridents use, that volume will then be encrypted upon trident installation.

Alternatively, encrypt the trident volume using the ONTAP CLI command `volume encryption conversion start -vserver SVM_name -volume volume_name`. Verify the status of the conversion operation using the command `volume encryption conversion show`. Please note that you cannot use *volume encryption conversion start* ONTAP CLI command to start encryption on a SnapLock or FlexGroup volume. For more information on how setup NetApp Volume Encryption, refer to the ONTAP NetApp Encryption Power Guide.

Element Backend

On the Solidfire backend, enable encryption for the cluster.

CHAPTER 4

Trident for Docker

Trident for Docker provides direct integration with the Docker ecosystem for NetApp's ONTAP, SolidFire, and E-Series storage platforms. It supports the provisioning and management of storage resources from the storage platform to Docker hosts, with a robust framework for adding additional platforms in the future.

Multiple instances of Trident can run concurrently on the same host. This allows simultaneous connections to multiple storage systems and storage types, with the ablity to customize the storage used for the Docker volume(s).

4.1 Deploying

1. Verify that your deployment meets all of the *requirements*.
2. Ensure that you have a supported version of Docker installed.

   ```
   docker --version
   ```

   If your version is out of date, follow the instructions for your distribution to install or update.
3. Verify that the protocol prerequisites are installed and configured on your host. See *Host Configuration*.
4. Create a configuration file. The default location is `/etc/netappdvp/config.json`. Be sure to use the correct options for your storage system.

   ```
   # create a location for the config files
   sudo mkdir -p /etc/netappdvp

   # create the configuration file, see below for more configuration examples
   cat << EOF > /etc/netappdvp/config.json
   {
       "version": 1,
       "storageDriverName": "ontap-nas",
       "managementLIF": "10.0.0.1",
       "dataLIF": "10.0.0.2",
       "svm": "svm_nfs",
   ```

(continues on next page)

(continued from previous page)

```
    "username": "vsadmin",
    "password": "secret",
    "aggregate": "aggr1"
}
EOF
```

5. Start Trident using the managed plugin system.

```
docker plugin install netapp/trident-plugin:19.01 --alias netapp --grant-all-
↪permissions
```

6. Begin using Trident to consume storage from the configured system.

```
# create a volume named "firstVolume"
docker volume create -d netapp --name firstVolume

# create a default volume at container instantiation
docker run --rm -it --volume-driver netapp --volume secondVolume:/my_vol alpine␣
↪ash

# remove the volume "firstVolume"
docker volume rm firstVolume
```

4.2 Host and storage configuration

4.2.1 Host Configuration

NFS

Install the following system packages:

- RHEL / CentOS

```
sudo yum install -y nfs-utils
```

- Ubuntu / Debian

```
sudo apt-get install -y nfs-common
```

iSCSI

- RHEL / CentOS
  1. Install the following system packages:

```
sudo yum install -y lsscsi iscsi-initiator-utils sg3_utils device-mapper-
↪multipath
```

  2. Start the multipathing daemon:

```
sudo mpathconf --enable --with_multipathd y
```

  3. Ensure that *iscsid* and *multipathd* are enabled and running:

```
sudo systemctl enable iscsid multipathd
sudo systemctl start iscsid multipathd
```

4. Discover the iSCSI targets:

```
sudo iscsiadm -m discoverydb -t st -p <DATA_LIF_IP> --discover
```

5. Login to the discovered iSCSI targets:

```
sudo iscsiadm -m node -p <DATA_LIF_IP> --login
```

6. Start and enable `iscsi`:

```
sudo systemctl enable iscsi
sudo systemctl start iscsi
```

- Ubuntu / Debian

1. Install the following system packages:

```
sudo apt-get install -y open-iscsi lsscsi sg3-utils multipath-tools scsitools
```

2. Enable multipathing:

```
sudo tee /etc/multipath.conf <<-'EOF'
defaults {
    user_friendly_names yes
    find_multipaths yes
}
EOF

sudo service multipath-tools restart
```

3. Ensure that `iscsid` and `multipathd` are running:

```
sudo service open-iscsi start
sudo service multipath-tools start
```

4. Discover the iSCSI targets:

```
sudo iscsiadm -m discoverydb -t st -p <DATA_LIF_IP> --discover
```

5. Login to the discovered iSCSI targets:

```
sudo iscsiadm -m node -p <DATA_LIF_IP> --login
```

Traditional Install Method (Docker <= 1.12)

1. Ensure you have Docker version 1.10 or above

```
docker --version
```

If your version is out of date, update to the latest.

```
curl -fsSL https://get.docker.com/ | sh
```

Or, follow the instructions for your distribution.

2. After ensuring the correct version of Docker is installed, install and configure the NetApp Docker Volume Plugin. Note, you will need to ensure that NFS and/or iSCSI is configured for your system. See the installation instructions below for detailed information on how to do this.

```
# download and unpack the application
wget https://github.com/NetApp/trident/releases/download/v19.01.0/trident-
↪installer-19.01.0.tar.gz
tar zxf trident-installer-19.01.0.tar.gz

# move to a location in the bin path
sudo mv trident-installer/extras/bin/trident /usr/local/bin
sudo chown root:root /usr/local/bin/trident
sudo chmod 755 /usr/local/bin/trident

# create a location for the config files
sudo mkdir -p /etc/netappdvp

# create the configuration file, see below for more configuration examples
cat << EOF > /etc/netappdvp/ontap-nas.json
{
    "version": 1,
    "storageDriverName": "ontap-nas",
    "managementLIF": "10.0.0.1",
    "dataLIF": "10.0.0.2",
    "svm": "svm_nfs",
    "username": "vsadmin",
    "password": "secret",
    "aggregate": "aggr1"
}
EOF
```

3. After placing the binary and creating the configuration file(s), start the Trident daemon using the desired configuration file.

   Note: Unless specified, the default name for the volume driver will be "netapp".

```
sudo trident --config=/etc/netappdvp/ontap-nas.json
```

4. Once the daemon is started, create and manage volumes using the Docker CLI interface.

```
docker volume create -d netapp --name trident_1
```

   Provision Docker volume when starting a container:

```
docker run --rm -it --volume-driver netapp --volume trident_2:/my_vol alpine ash
```

   Destroy docker volume:

```
docker volume rm trident_1
docker volume rm trident_2
```

Starting Trident at System Startup

A sample unit file for systemd based systems can be found at `contrib/trident.service.example` in the git repo. To use the file, with CentOS/RHEL:

```
# copy the file to the correct location.  you must use unique names for the
# unit files if you have more than one instance running
cp contrib/trident.service.example /usr/lib/systemd/system/trident.service

# edit the file, change the description (line 2) to match the driver name and the
# configuration file path (line 9) to reflect your environment.

# reload systemd for it to ingest changes
systemctl daemon-reload

# enable the service, note this name will change depending on what you named the
# file in the /usr/lib/systemd/system directory
systemctl enable trident

# start the service, see note above about service name
systemctl start trident

# view the status
systemctl status trident
```

Note that anytime the unit file is modified you will need to issue the command `systemctl daemon-reload` for it to be aware of the changes.

Docker Managed Plugin Method (Docker >= 1.13 / 17.03)

Note: If you have used Trident pre-1.13/17.03 in the traditional daemon method, please ensure that you stop the Trident process and restart your Docker daemon before using the managed plugin method.

```
# stop all running instances
pkill /usr/local/bin/netappdvp
pkill /usr/local/bin/trident

# restart docker
systemctl restart docker
```

Trident Specific Plugin Startup Options

- `config` - Specify the configuration file the plugin will use. Only the file name should be specified, e.g. `gold.json`, the location must be `/etc/netappdvp` on the host system. The default is `config.json`.
- `log-level` - Specify the logging level (`debug`, `info`, `warn`, `error`, `fatal`). The default is `info`.
- `debug` - Specify whether debug logging is enabled. Default is false. Overrides log-level if true.

Installing the Managed Plugin

1. Ensure you have Docker Engine 17.03 (nee 1.13) or above installed.

   ```
   docker --version
   ```

   If your version is out of date, follow the instructions for your distribution to install or update.

2. Create a configuration file. The config file must be located in the `/etc/netappdvp` directory. The default filename is `config.json`, however you can use any name you choose by specifying the `config` option with the file name. Be sure to use the correct options for your storage system.

   ```
   # create a location for the config files
   sudo mkdir -p /etc/netappdvp
   ```

(continues on next page)

(continued from previous page)

```
# create the configuration file, see below for more configuration examples
cat << EOF > /etc/netappdvp/config.json
{
    "version": 1,
    "storageDriverName": "ontap-nas",
    "managementLIF": "10.0.0.1",
    "dataLIF": "10.0.0.2",
    "svm": "svm_nfs",
    "username": "vsadmin",
    "password": "secret",
    "aggregate": "aggr1"
}
EOF
```

3. Start Trident using the managed plugin system.

```
docker plugin install --grant-all-permissions --alias netapp netapp/trident-
↪plugin:19.01 config=myConfigFile.json
```

4. Begin using Trident to consume storage from the configured system.

```
# create a volume named "firstVolume"
docker volume create -d netapp --name firstVolume

# create a default volume at container instantiation
docker run --rm -it --volume-driver netapp --volume secondVolume:/my_vol alpine␣
↪ash

# remove the volume "firstVolume"
docker volume rm firstVolume
```

4.2.2 Global Configuration

These configuration variables apply to all Trident configurations, regardless of the storage platform being used.

Option	Description	Example
`version`	Config file version number	1
`storageDriverName`	`ontap-nas`, `ontap-nas-economy`, `ontap-nas-flexgroup`, `ontap-san`, `eseries-iscsi`, or `solidfire-san`	ontap-nas
`storagePrefix`	Optional prefix for volume names. Default: “netappdvp_”	netappdvp_
`limitVolumeSize`	Optional restriction on volume sizes. Default: “” (not enforced)	10g

Also, default option settings are available to avoid having to specify them on every volume create. The `size` option is available for all controller types. See the ONTAP config section for an example of how to set the default volume size.

Defaults Option	Description	Example
`size`	Optional default size for new volumes. Default: “1G”	10G

Storage Prefix

A new config file variable has been added in v1.2 called "storagePrefix" that allows you to modify the prefix applied to volume names by the plugin. By default, when you run *docker volume create*, the volume name supplied is prepended with "netappdvp_" *("netappdvp-" for SolidFire)*.

If you wish to use a different prefix, you can specify it with this directive. Alternatively, you can use *pre-existing* volumes with the volume plugin by setting `storagePrefix` to an empty string, "".

SolidFire specific recommendation do not use a storagePrefix (including the default). By default the SolidFire driver will ignore this setting and not use a prefix. We recommend using either a specific tenantID for docker volume mapping or using the attribute data which is populated with the docker version, driver info and raw name from docker in cases where any name munging may have been used.

A note of caution: *docker volume rm* will *delete* these volumes just as it does volumes created by the plugin using the default prefix. Be very careful when using pre-existing volumes!

4.2.3 ONTAP Configuration

User Permissions

Trident does not need full permissions on the ONTAP cluster and should not be used with the cluster-level admin account. Below are the ONTAP CLI comands to create a dedicated user for Trident with specific permissions.

```
# create a new Trident role
security login role create -vserver [VSERVER] -role trident_role -cmddirname DEFAULT -
↪access none

# grant common Trident permissions
security login role create -vserver [VSERVER] -role trident_role -cmddirname "event␣
↪generate-autosupport-log" -access all
security login role create -vserver [VSERVER] -role trident_role -cmddirname "network␣
↪interface" -access readonly
security login role create -vserver [VSERVER] -role trident_role -cmddirname "version
↪" -access readonly
security login role create -vserver [VSERVER] -role trident_role -cmddirname "vserver
↪" -access readonly
security login role create -vserver [VSERVER] -role trident_role -cmddirname "vserver␣
↪nfs show" -access readonly
security login role create -vserver [VSERVER] -role trident_role -cmddirname "volume"␣
↪-access all
security login role create -vserver [VSERVER] -role trident_role -cmddirname
↪"snapmirror" -access all

# grant ontap-san Trident permissions
security login role create -vserver [VSERVER] -role trident_role -cmddirname "vserver␣
↪iscsi show" -access readonly
security login role create -vserver [VSERVER] -role trident_role -cmddirname "lun" -
↪access all

# grant ontap-nas-economy Trident permissions
security login role create -vserver [VSERVER] -role trident_role -cmddirname "vserver␣
↪export-policy create" -access all
security login role create -vserver [VSERVER] -role trident_role -cmddirname "vserver␣
↪export-policy rule create" -access all

# create a new Trident user with Trident role
security login create -vserver [VSERVER] -username trident_user -role trident_role -
↪application ontapi -authmethod password
```

Configuration File Options

In addition to the global configuration values above, when using ONTAP these top level options are available.

Option	Description	Example
managementLIF	IP address of ONTAP management LIF	10.0.0.1
dataLIF	IP address of protocol LIF; will be derived if not specified	10.0.0.2
svm	Storage virtual machine to use (req, if management LIF is a cluster LIF)	svm_nfs
username	Username to connect to the storage device	vsadmin
password	Password to connect to the storage device	secret
aggregate	Aggregate for provisioning (optional; if set, must be assigned to the SVM)	aggr1
limitAggregateUsage	Fail provisioning if usage is above this percentage (optional)	75%

A fully-qualified domain name (FQDN) can be specified for the managementLIF option. For the ontap-nas* drivers only, a FQDN may also be specified for the dataLIF option, in which case the FQDN will be used for the NFS mount operations. For the ontap-san driver, the default is to use all data LIF IPs from the SVM and to use iSCSI multipath. Specifying an IP address for the dataLIF for the ontap-san driver forces the driver to disable multipath and use only the specified address.

For the ontap-nas and ontap-nas-economy drivers, an additional top level option is available. For NFS host configuration, see also: http://www.netapp.com/us/media/tr-4067.pdf

For the ontap-nas-flexgroup driver, the `aggregate` option in the configuration file is ignored. All aggregates assigned to the SVM are used to provision a FlexGroup Volume.

Option	Description	Example
nfsMountOptions	Fine grained control of NFS mount options; defaults to "-o nfsvers=3"	-o nfsvers=4

For the ontap-san driver, an additional top level option is available to specify an igroup.

Option	Description	Example
igroupName	The igroup used by the plugin; defaults to "netappdvp"	myigroup

For the ontap-nas-economy driver, the `limitVolumeSize` option will additionally limit the size of the FlexVols that it creates.

Option	Description	Example
limitVolumeSize	Maximum requestable volume size and qtree parent volume size	300g

Also, when using ONTAP, these default option settings are available to avoid having to specify them on every volume create.

Defaults Option	Description	Example
spaceReserve	Space reservation mode; “none” (thin provisioned) or “volume” (thick)	none
snapshotPolicy	Snapshot policy to use, default is “none”	none
snapshotReserve	Snapshot reserve percentage, default is “” to accept ONTAP’s default	10
splitOnClone	Split a clone from its parent upon creation, defaults to “false”	false
encryption	Enable NetApp Volume Encryption, defaults to “false”	true
unixPermissions	NAS option for provisioned NFS volumes, defaults to “777”	777
snapshotDir	NAS option for access to the .snapshot directory, defaults to “false”	false
exportPolicy	NAS option for the NFS export policy to use, defaults to “default”	default
securityStyle	NAS option for access to the provisioned NFS volume, defaults to “unix”	mixed
fileSystemType	SAN option to select the file system type, defaults to “ext4”	xfs

Scaling Options

The ontap-nas and ontap-san drivers create an ONTAP FlexVol for each Docker volume. ONTAP supports up to 1000 FlexVols per cluster node with a cluster maximum of 12,000 FlexVols. If your Docker volume requirements fit within that limitation, the ontap-nas driver is the preferred NAS solution due to the additional features offered by FlexVols such as Docker-volume-granular snapshots and cloning.

If you need more Docker volumes than may be accommodated by the FlexVol limits, choose the ontap-nas-economy driver, which creates Docker volumes as ONTAP Qtrees within a pool of automatically managed FlexVols. Qtrees offer far greater scaling, up to 100,000 per cluster node and 2,400,000 per cluster, at the expense of some features. The ontap-nas-economy driver does not support Docker-volume-granular snapshots or cloning. The ontap-nas-economy driver is not currently supported in Docker Swarm, as Swarm does not orchestrate volume creation across multiple nodes.

Choose the ontap-nas-flexgroup driver to increase parallelism to a single volume that can grow into the petabyte range with billions of files. Some ideal use cases for FlexGroups include AI/ML/DL, big data and analytics, software builds, streaming, file repositories, etc. Trident uses all aggregates assigned to an SVM when provisioning a FlexGroup Volume. FlexGroup support in Trident also has the following considerations:

- Requires ONTAP version 9.2 or greater.
- As of this writing, FlexGroups only support NFS v3.
- Recommended to enable the 64-bit NFSv3 identifiers for the SVM.
- The minimum recommended FlexGroup size is 100GB.
- Cloning is not supported for FlexGroup Volumes.

For information regarding FlexGroups and workloads that are appropriate for FlexGroups see the NetApp FlexGroup Volume - Best Practices and Implementation Guide.

To get advanced features and huge scale in the same environment, you can run multiple instances of the Docker Volume Plugin, with one using ontap-nas and another using ontap-nas-economy.

Example ONTAP Config Files

NFS Example for ontap-nas driver

```
{
    "version": 1,
    "storageDriverName": "ontap-nas",
    "managementLIF": "10.0.0.1",
```

(continues on next page)

(continued from previous page)

```
    "dataLIF": "10.0.0.2",
    "svm": "svm_nfs",
    "username": "vsadmin",
    "password": "secret",
    "aggregate": "aggr1",
    "defaults": {
      "size": "10G",
      "spaceReserve": "none",
      "exportPolicy": "default"
    }
}
```

NFS Example for ontap-nas-flexgroup driver

```
{
    "version": 1,
    "storageDriverName": "ontap-nas-flexgroup",
    "managementLIF": "10.0.0.1",
    "dataLIF": "10.0.0.2",
    "svm": "svm_nfs",
    "username": "vsadmin",
    "password": "secret",
    "defaults": {
      "size": "100G",
      "spaceReserve": "none",
      "exportPolicy": "default"
    }
}
```

NFS Example for ontap-nas-economy driver

```
{
    "version": 1,
    "storageDriverName": "ontap-nas-economy",
    "managementLIF": "10.0.0.1",
    "dataLIF": "10.0.0.2",
    "svm": "svm_nfs",
    "username": "vsadmin",
    "password": "secret",
    "aggregate": "aggr1"
}
```

iSCSI Example for ontap-san driver

```
{
    "version": 1,
    "storageDriverName": "ontap-san",
    "managementLIF": "10.0.0.1",
    "dataLIF": "10.0.0.3",
    "svm": "svm_iscsi",
    "username": "vsadmin",
    "password": "secret",
    "aggregate": "aggr1",
    "igroupName": "myigroup"
}
```

4.2.4 SolidFire Configuration

In addition to the global configuration values above, when using SolidFire, these options are available.

Option	Description	Example
`Endpoint`	Ex. `https://<login>:<password>@<mvip>/json-rpc/<element-version>`	
`SVIP`	iSCSI IP address and port	10.0.0.7:3260
`TenantName`	SF Tenant to use (created if not found)	"docker"
`InitiatorIFace`	Specify interface when restricting iSCSI traffic to non-default interface	"default"
`Types`	QoS specifications	See below
`LegacyNamePrefix`	Prefix for upgraded Trident installs	"netappdvp-"

The SolidFire driver does not support Docker Swarm.

LegacyNamePrefix If you used a version of Trident prior to 1.3.2 and perform an upgrade with existing volumes, you'll need to set this value in order to access your old volumes that were mapped via the `volume-name` method.

Example Solidfire Config File

```
{
    "version": 1,
    "storageDriverName": "solidfire-san",
    "Endpoint": "https://admin:admin@192.168.160.3/json-rpc/8.0",
    "SVIP": "10.0.0.7:3260",
    "TenantName": "docker",
    "InitiatorIFace": "default",
    "Types": [
        {
            "Type": "Bronze",
            "Qos": {
                "minIOPS": 1000,
                "maxIOPS": 2000,
                "burstIOPS": 4000
            }
        },
        {
            "Type": "Silver",
            "Qos": {
                "minIOPS": 4000,
                "maxIOPS": 6000,
                "burstIOPS": 8000
            }
        },
        {
            "Type": "Gold",
            "Qos": {
                "minIOPS": 6000,
                "maxIOPS": 8000,
                "burstIOPS": 10000
            }
        }
    ]
}
```

4.2.5 E-Series Configuration

In addition to the global configuration values above, when using E-Series, these options are available.

Option	Description	Example
webProxyHostname	Hostname or IP address of Web Services Proxy	localhost
webProxyPort	Port number of the Web Services Proxy (optional)	8443
webProxyUseHTTP	Use HTTP instead of HTTPS for Web Services Proxy (default = false)	true
webProxyVerifyTLS	Verify server's certificate chain and hostname (default = false)	true
username	Username for Web Services Proxy	rw
password	Password for Web Services Proxy	rw
controllerA	IP address of controller A	10.0.0.5
controllerB	IP address of controller B	10.0.0.6
passwordArray	Password for storage array if set	blank/empty
hostDataIP	Host iSCSI IP address (if multipathing just choose either one)	10.0.0.101
poolNameSearchPattern	Regular expression for matching storage pools available for Trident volumes (default = .+)	docker.*
hostType	Type of E-series Host created by Trident (default = linux_dm_mp)	linux_dm_mp
accessGroupName	Name of E-series Host Group to contain Hosts defined by Trident (default = netappdvp)	Docker-Hosts

Example E-Series Config File

Example for eseries-iscsi driver

```
{
  "version": 1,
  "storageDriverName": "eseries-iscsi",
  "webProxyHostname": "localhost",
  "webProxyPort": "8443",
  "webProxyUseHTTP": false,
  "webProxyVerifyTLS": true,
  "username": "rw",
  "password": "rw",
  "controllerA": "10.0.0.5",
  "controllerB": "10.0.0.6",
  "passwordArray": "",
  "hostDataIP": "10.0.0.101"
}
```

E-Series Array Setup Notes

The E-Series Docker driver can provision Docker volumes in any storage pool on the array, including volume groups and DDP pools. To limit the Docker driver to a subset of the storage pools, set the `poolNameSearchPattern` in the configuration file to a regular expression that matches the desired pools.

When creating a docker volume you can specify the volume size as well as the disk media type using the `-o` option and the tags `size` and `mediaType`. Valid values for media type are `hdd` and `ssd`. Note that these are optional; if unspecified, the defaults will be a *1 GB* volume allocated from an *HDD pool*. An example of using these tags to create a 2 GiB volume from an SSD-based pool:

```
docker volume create -d netapp --name my_vol -o size=2G -o mediaType=ssd
```

The E-series Docker driver will detect and use any preexisting Host definitions without modification, and the driver will automatically define Host and Host Group objects as needed. The host type for hosts created by the driver defaults to `linux_dm_mp`, the native DM-MPIO multipath driver in Linux.

The current E-series Docker driver only supports iSCSI.

4.2.6 Multiple Instances of Trident

Multiple instances of Trident are needed when you desire to have multiple storage configurations available simultaneously. The key to multiple instances is to give them different names using the `--alias` option with the containerized plugin, or `--volume-driver` option when instantiating Trident on the host.

Docker Managed Plugin (Docker >= 1.13 / 17.03)

1. Launch the first instance specifying an alias and configuration file

```
docker plugin install --grant-all-permissions --alias silver netapp/trident-
↪plugin:19.01 config=silver.json
```

2. Launch the second instance, specifying a different alias and configuration file

```
docker plugin install --grant-all-permissions --alias gold netapp/trident-
↪plugin:19.01 config=gold.json
```

3. Create volumes specifying the alias as the driver name

```
# gold volume
docker volume create -d gold --name ntapGold

# silver volume
docker volume create -d silver --name ntapSilver
```

Traditional (Docker <=1.12)

1. Launch the plugin with an NFS configuration using a custom driver ID:

```
sudo trident --volume-driver=netapp-nas --config=/path/to/config-nfs.json
```

2. Launch the plugin with an iSCSI configuration using a custom driver ID:

```
sudo trident --volume-driver=netapp-san --config=/path/to/config-iscsi.
↪json
```

3. Provision Docker volumes each driver instance:

   - NFS

```
docker volume create -d netapp-nas --name my_nfs_vol
```

   - iSCSI

```
docker volume create -d netapp-san --name my_iscsi_vol
```

4.3 Common tasks

4.3.1 Managing Trident

Installing Trident

Follow the extensive *deployment* guide.

Updating Trident

The plugin is not in the data path, therefore you can safely upgrade it without any impact to volumes that are in use. As with any plugin, during the upgrade process there will be a brief period where 'docker volume' commands directed at the plugin will not succeed, and applications will be unable to mount volumes until the plugin is running again. Under most circumstances, this is a matter of seconds.

1. List the existing volumes:

```
docker volume ls
DRIVER              VOLUME NAME
netapp:latest       my_volume
```

2. Disable the plugin:

```
docker plugin disable -f netapp:latest
docker plugin ls
ID                  NAME                DESCRIPTION                                                 ␣
↪ENABLED
7067f39a5df5        netapp:latest       nDVP - NetApp Docker Volume Plugin    false
```

3. Upgrade the plugin:

```
docker plugin upgrade --skip-remote-check --grant-all-permissions netapp:latest␣
↪netapp/trident-plugin:19.01
```

> **Note:** The 18.01 release of Trident replaces the nDVP. You should upgrade directly from the netapp/ndvp-plugin image to the netapp/trident-plugin image.

4. Enable the plugin:

```
docker plugin enable netapp:latest
```

5. Verify that the plugin is enabled:

```
docker plugin ls
ID                  NAME                DESCRIPTION                                                 ␣
↪ENABLED
7067f39a5df5        netapp:latest       Trident - NetApp Docker Volume Plugin                       ␣
↪true
```

6. Verify that the volumes are visible:

```
docker volume ls
DRIVER              VOLUME NAME
netapp:latest       my_volume
```

Uninstalling Trident

> **Note:** Do not uninstall the plugin in order to upgrade it, as Docker may become confused as to which plugin owns any existing volumes; use the upgrade instructions in the previous section instead.

1. Remove any volumes that the plugin created.
2. Disable the plugin:

```
docker plugin disable netapp:latest
docker plugin ls
ID                  NAME                DESCRIPTION                                ␣
↪ENABLED
7067f39a5df5        netapp:latest       nDVP - NetApp Docker Volume Plugin    false
```

3. Remove the plugin:

```
docker plugin rm netapp:latest
```

4.3.2 Managing volumes

Creating and consuming storage from ONTAP, SolidFire, and/or E-Series systems is easy with Trident. Simply use the standard `docker volume` commands with the nDVP driver name specified when needed.

Create a Volume

```
# create a volume with a Trident driver using the default name
docker volume create -d netapp --name firstVolume

# create a volume with a specific Trident instance
docker volume create -d ntap_bronze --name bronzeVolume
```

If no options are specified, the defaults for the driver are used. The defaults are documented on the page for the storage driver you're using below.

The default volume size may be overridden per volume as follows:

```
# create a 20GiB volume with a Trident driver
docker volume create -d netapp --name my_vol --opt size=20G
```

Volume sizes are expressed as strings containing an integer value with optional units (e.g. "10G", "20GB", "3TiB"). If no units are specified, the default is 'G'. Size units may be expressed either as powers of 2 (B, KiB, MiB, GiB, TiB) or powers of 10 (B, KB, MB, GB, TB). Shorthand units use powers of 2 (G = GiB, T = TiB, . . .).

If the snapshot reserve is not specified and the snapshot policy is 'none', Trident will use a snapshot reserve of 0%.

```
# create a volume with no snapshot policy and no snapshot reserve
docker volume create -d netapp --name my_vol --opt snapshotPolicy=none

# create a volume with no snapshot policy and a custom snapshot reserve of 10%
docker volume create -d netapp --name my_vol --opt snapshotPolicy=none --opt␣
↪snapshotReserve=10
```

(continues on next page)

(continued from previous page)

```
# create a volume with a snapshot policy and a custom snapshot reserve of 10%
docker volume create -d netapp --name my_vol --opt snapshotPolicy=myPolicy --opt␣
↪snapshotReserve=10

# create a volume with a snapshot policy, and accept ONTAP's default snapshot reserve␣
↪(usually 5%)
docker volume create -d netapp --name my_vol --opt snapshotPolicy=myPolicy
```

Volume Driver CLI Options

Each storage driver has a different set of options which can be provided at volume creation time to customize the outcome. Refer to the documentation below for your configured storage system to determine which options apply.

ONTAP Volume Options

Volume create options for both NFS and iSCSI:

- `size` - the size of the volume, defaults to 1 GiB
- `spaceReserve` - thin or thick provision the volume, defaults to thin. Valid values are `none` (thin provisioned) and `volume` (thick provisioned).
- `snapshotPolicy` - this will set the snapshot policy to the desired value. The default is `none`, meaning no snapshots will automatically be created for the volume. Unless modified by your storage administrator, a policy named "default" exists on all ONTAP systems which creates and retains six hourly, two daily, and two weekly snapshots. The data preserved in a snapshot can be recovered by browsing to the .snapshot directory in any directory in the volume.
- `snapshotReserve` - this will set the snapshot reserve to the desired percentage. The default is no value, meaning ONTAP will select the snapshotReserve (usually 5%) if you have selected a snapshotPolicy, or 0% if the snapshotPolicy is `none`. The default snapshotReserve value may be set in the config file for all ONTAP backends, and it may be used as a volume creation option for all ONTAP backends except ontap-nas-economy.
- `splitOnClone` - when cloning a volume, this will cause ONTAP to immediately split the clone from its parent. The default is `false`. Some use cases for cloning volumes are best served by splitting the clone from its parent immediately upon creation, since there is unlikely to be any opportunity for storage efficiencies. For example, cloning an empty database can offer large time savings but little storage savings, so it's best to split the clone immediately.
- `encryption` - this will enable NetApp Volume Encryption (NVE) on the new volume, defaults to `false`. NVE must be licensed and enabled on the cluster to use this option.

NFS has additional options that aren't relevant when using iSCSI:

- `unixPermissions` - this controls the permission set for the volume itself. By default the permissions will be set to `---rwxr-xr-x`, or in numerical notation `0755`, and root will be the owner. Either the text or numerical format will work.
- `snapshotDir` - setting this to `true` will make the .snapshot directory visible to clients accessing the volume. The default value is `false`, meaning that access to snapshot data is disabled by default. Some images, for example the official MySQL image, don't function as expected when the .snapshot directory is visible.
- `exportPolicy` - sets the export policy to be used for the volume. The default is `default`.
- `securityStyle` - sets the security style to be used for access to the volume. The default is `unix`. Valid values are `unix` and `mixed`.

iSCSI has an additional option that isn't relevant when using NFS:

- `fileSystemType` - sets the file system used to format iSCSI volumes. The default is `ext4`. Valid values are `ext3`, `ext4`, and `xfs`.

Using these options during the docker volume create operation is super simple, just provide the option and the value using the `-o` operator during the CLI operation. These override any equivalent vales from the JSON configuration file.

```
# create a 10GiB volume
docker volume create -d netapp --name demo -o size=10G -o encryption=true

# create a 100GiB volume with snapshots
docker volume create -d netapp --name demo -o size=100G -o snapshotPolicy=default -o␣
↪snapshotReserve=10

# create a volume which has the setUID bit enabled
docker volume create -d netapp --name demo -o unixPermissions=4755
```

The minimum volume size is 20MiB.

SolidFire Volume Options

The SolidFire driver options expose the size and quality of service (QoS) policies associated with the volume. When the volume is created, the QoS policy associated with it is specified using the `-o type=service_level` nomenclature.

The first step to defining a QoS service level with the SolidFire driver is to create at least one type and specify the minimum, maximum, and burst IOPS associated with a name in the configuration file.

Example SolidFire Configuration File with QoS Definitions

```
{
    "...": "..."
    "Types": [
        {
            "Type": "Bronze",
            "Qos": {
                "minIOPS": 1000,
                "maxIOPS": 2000,
                "burstIOPS": 4000
            }
        },
        {
            "Type": "Silver",
            "Qos": {
                "minIOPS": 4000,
                "maxIOPS": 6000,
                "burstIOPS": 8000
            }
        },
        {
            "Type": "Gold",
            "Qos": {
                "minIOPS": 6000,
                "maxIOPS": 8000,
                "burstIOPS": 10000
            }
        }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

In the above configuration we have three policy definitions: *Bronze*, *Silver*, and *Gold*. These names are well known and fairly common, but we could have just as easily chosen: *pig*, *horse*, and *cow*, the names are arbitrary.

```
# create a 10GiB Gold volume
docker volume create -d solidfire --name sfGold -o type=Gold -o size=10G

# create a 100GiB Bronze volume
docker volume create -d solidfire --name sfBronze -o type=Bronze -o size=100G
```

Other SolidFire Create Options

Volume create options for SolidFire:

- `size` - the size of the volume, defaults to 1GiB or config entry `... "defaults": {"size": "5G"}`
- `blocksize` - use either `512` or `4096`, defaults to 512 or config entry `DefaultBlockSize`

E-Series Volume Options

Media Type

The E-Series driver offers the ability to specify the type of disk which will be used to back the volume and, like the other drivers, the ability to set the size of the volume at creation time.

Currently only two values for `mediaType` are supported: `ssd` and `hdd`.

```
# create a 10GiB SSD backed volume
docker volume create -d eseries --name eseriesSsd -o mediaType=ssd -o size=10G

# create a 100GiB HDD backed volume
docker volume create -d eseries --name eseriesHdd -o mediaType=hdd -o size=100G
```

File System Type

The user can specify the file system type to use to format the volume. The default for `fileSystemType` is `ext4`. Valid values are `ext3`, `ext4`, and `xfs`.

```
# create a volume using xfs
docker volume create -d eseries --name xfsVolume -o fileSystemType=xfs
```

Destroy a Volume

```
# destroy the volume just like any other Docker volume
docker volume rm firstVolume
```

Volume Cloning

When using the ontap-nas, ontap-san, and solidfire-san storage drivers, the Docker Volume Plugin can clone volumes. When using the ontap-nas-flexgroup or ontap-nas-economy drivers, cloning is not supported.

```
# inspect the volume to enumerate snapshots
docker volume inspect <volume_name>

# create a new volume from an existing volume.  this will result in a new snapshot␣
↪being created
docker volume create -d <driver_name> --name <new_name> -o from=<source_docker_volume>

# create a new volume from an existing snapshot on a volume.  this will not create a␣
↪new snapshot
docker volume create -d <driver_name> --name <new_name> -o from=<source_docker_volume>
↪ -o fromSnapshot=<source_snap_name>
```

Here is an example of that in action:

```
[me@host ~]$ docker volume inspect firstVolume

[
    {
        "Driver": "ontap-nas",
        "Labels": null,
        "Mountpoint": "/var/lib/docker-volumes/ontap-nas/netappdvp_firstVolume",
        "Name": "firstVolume",
        "Options": {},
        "Scope": "global",
        "Status": {
            "Snapshots": [
                {
                    "Created": "2017-02-10T19:05:00Z",
                    "Name": "hourly.2017-02-10_1505"
                }
            ]
        }
    }
]

[me@host ~]$ docker volume create -d ontap-nas --name clonedVolume -o from=firstVolume
clonedVolume

[me@host ~]$ docker volume rm clonedVolume
[me@host ~]$ docker volume create -d ontap-nas --name volFromSnap -o from=firstVolume␣
↪-o fromSnapshot=hourly.2017-02-10_1505
volFromSnap

[me@host ~]$ docker volume rm volFromSnap
```

Access Externally Created Volumes

Externally created block devices (or their clones) may be accessed by containers using Trident only if they have no partitions and if their filesystem is supported by nDVP (example: an ext4-formatted /dev/sdc1 will not be accessible via nDVP).

4.4 Known issues

1. Volume names must be a minimum of 2 characters in length

   Note: This is a Docker client limitation. The client will interpret a single character name as being a Windows path. See bug 25773.

2. Docker Swarm has certain behaviors that prevent us from supporting it with every storage and driver combination:

   - Docker Swarm presently makes use of volume name instead of volume ID as its unique volume identifier.
   - Volume requests are simultaneously sent to each node in a Swarm cluster.
   - Volume Plugins (including Trident) must run independently on each node in a Swarm cluster.

   Due to the way ONTAP works and how the ontap-nas and ontap-san drivers function, they are the only ones that happen to be able to operate within these limitations.

   The rest of the drivers are subject to issues like race conditions that can result in the creation of a large number of volumes for a single request without a clear "winner"; for example, Element has a feature that allows volumes to have the same name but different IDs.

   NetApp has provided feedback to the Docker team, but does not have any indication of future recourse.

3. If a FlexGroup is in the process of being provisioned, ONTAP will not provision a second FlexGroup if the second FlexGroup has one or more aggregates in common with the FlexGroup being provisioned.

4.5 Troubleshooting

The most common problem new users run into is a misconfiguration that prevents the plugin from initializing. When this happens you will likely see a message like this when you try to install or enable the plugin:

```
Error response from daemon: dial unix /run/docker/plugins/<id>/netapp.sock:␣
↪connect: no such file or directory
```

This simply means that the plugin failed to start. Luckily, the plugin has been built with a comprehensive logging capability that should help you diagnose most of the issues you are likely to come across.

The method you use to access or tune those logs varies based on how you are running the plugin.

4.5.1 Managed plugin method

If you are running Trident using the recommended managed plugin method (i.e., using `docker plugin` commands), the logs are passed through Docker itself, so they will be interleaved with Docker's own logging.

To view them, simply run:

```
# docker plugin ls
ID                  NAME                DESCRIPTION                          ␣
↪ENABLED
4fb97d2b956b        netapp:latest       nDVP - NetApp Docker Volume Plugin   ␣
↪false

# journalctl -u docker | grep 4fb97d2b956b
```

The standard logging level should allow you to diagnose most issues. If you find that's not enough, you can enable debug logging:

```
# install the plugin with debug logging enabled
docker plugin install netapp/trident-plugin:<version> --alias <alias>␣
↪debug=true

# or, enable debug logging when the plugin is already installed
docker plugin disable <plugin>
docker plugin set <plugin> debug=true
docker plugin enable <plugin>
```

4.5.2 Binary method

If you are not running as a managed plugin, you are running the binary itself on the host. The logs are available in the host's `/var/log/netappdvp` directory. If you need to enable debug logging, specify `-debug` when you run the plugin.

CHAPTER 5

Requirements

5.1 Supported frontends (orchestrators)

Trident supports multiple container engines and orchestrators, including:

- Docker 17.06 (CE or EE) or later (latest: 18.09)
- Docker Enterprise Edition 17.06 or later (latest: 2.1)
- Kubernetes 1.8 or later (latest: 1.13)
- OpenShift 3.8 or later (latest: 3.11)

In addition, Trident should work with any distribution of Docker or Kubernetes that uses one of the supported versions as a base, such as Rancher or Tectonic.

5.2 Supported backends (storage)

To use Trident, you need one or more of the following supported backends:

- FAS/AFF/Select/Cloud ONTAP 8.3 or later
- SolidFire Element OS 8 or later
- E/EF-Series SANtricity

5.3 Supported host operating systems

By default Trident itself runs in a container, therefore it will run on any Linux worker.

However, those workers do need to be able to mount the volumes that Trident provides using the standard NFS client or iSCSI initiator, depending on the backend(s) you're using.

These are the Linux distributions that are known to work:

- Debian 8 or later
- Ubuntu 16.04 or later
- CentOS 7.0 or later
- RHEL 7.0 or later
- CoreOS 1353.8.0 or later

The `tridentctl` utility also runs on any of these distributions of Linux.

5.4 Host configuration

Depending on the backend(s) in use, NFS and/or iSCSI utilities must be installed on all of the workers in the cluster. See the *worker preparation* guide for details.

5.5 Storage system configuration

Trident may require some changes to a storage system before a backend configuration can use it. See the *backend configuration* guide for details.

5.6 External etcd cluster (Optional)

Trident uses etcd v3.1.3 or later to store its metadata. The standard installation process includes an etcd container that is managed by Trident and backed by a volume from a supported storage system, so there is no need to install it separately.

If you would prefer to use a separate external etcd cluster instead, Trident can easily be configured to do so. See the *external etcd guide* for details.

CHAPTER 6

Getting help

Trident is an officially supported NetApp project. That means you can reach out to NetApp using any of the standard mechanisms and get the enterprise grade support that you need.

There is also a vibrant public community of container users (including Trident developers) on the #containers channel in NetApp's Slack team. This is a great place to ask general questions about the project and discuss related topics with like-minded peers.

CHAPTER 7

trident

Trident exposes several command-line options. Normally the defaults will suffice, but you may want to modify them in your deployment. They are:

7.1 Logging

- `-debug`: Optional; enables debugging output.
- `-loglevel <level>`: Optional; sets the logging level (debug, info, warn, error, fatal). Defaults to info.

7.2 Persistence

- `-etcd_v3 <address> or -etcd_v2 <address>`: Required; use this to specify the etcd deployment that Trident should use.
- `-etcd_v3_cert <file>`: Optional, etcdV3 client certificate.
- `-etcd_v3_cacert <file>`: Optional, etcdV3 client CA certificate.
- `-etcd_v3_key <file>`: Optional, etcdV3 client private key.
- `-no_persistence`: Optional, does not persist any metadata at all.
- `-passthrough`: Optional, uses backend as the sole source of truth.

7.3 Kubernetes

- `-k8s_pod`: Optional; however, either this or -k8s_api_server must be set to enable Kubernetes support. Setting this will cause 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>`: Optional; however, either this or -k8s_pod must be used to enable Kubernetes support. When specified, Trident will connect to the Kubernetes API server using the provided insecure address and port. This allows 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.
- `-k8s_config_path <file>`: Optional; path to a KubeConfig file.

7.4 Docker

- `-volume_driver <name>`: Optional; driver name used when registering the Docker plugin. Defaults to 'netapp'.
- `-driver_port <port-number>`: Optional; listen on this port rather than a UNIX domain socket.
- `-config <file>`: Path to a backend configuration file.

7.5 REST

- `-address <ip-or-host>`: Optional; 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 will not be directly accessible from outside the pod. Use -address "" to make the REST interface accessible from the pod IP address.
- `-port <port-number>`: Optional; specifies the port on which Trident's REST server should listen. Defaults to 8000.
- `-rest`: Optional; enable the REST interface. Defaults to true.

CHAPTER 8

tridentctl

The Trident installer bundle includes a command-line utility, `tridentctl`, that provides simple access to Trident. It can be used to install Trident, as well as to interact with it directly by any Kubernetes users with sufficient privileges, to manage the namespace that contains the Trident pod.

For full usage information, run `tridentctl --help`. Here are the available commands and global options:

```
Usage:
  tridentctl [command]

Available Commands:
  create      Add a resource to Trident
  delete      Remove one or more resources from Trident
  get         Get one or more resources from Trident
  help        Help about any command
  install     Install Trident
  logs        Print the logs from Trident
  uninstall   Uninstall Trident
  update      Modify a resource in Trident
  version     Print the version of Trident

Flags:
  -d, --debug              Debug output
  -h, --help               help for tridentctl
  -n, --namespace string   Namespace of Trident deployment
  -o, --output string      Output format. One of json|yaml|name|wide|ps (default)
  -s, --server string      Address/port of Trident REST interface
```

8.1 create

Add a resource to Trident

```
Usage:
  tridentctl create [command]

Available Commands:
  backend     Add a backend to Trident
```

8.2 delete

Remove one or more resources from Trident

```
Usage:
  tridentctl delete [command]

Available Commands:
  backend      Delete one or more storage backends from Trident
  storageclass Delete one or more storage classes from Trident
  volume       Delete one or more storage volumes from Trident
```

8.3 get

Get one or more resources from Trident

```
Usage:
  tridentctl get [command]

Available Commands:
  backend      Get one or more storage backends from Trident
  storageclass Get one or more storage classes from Trident
  volume       Get one or more volumes from Trident
```

8.4 install

Install Trident

```
Usage:
  tridentctl install [flags]

Flags:
      --dry-run                  Run all the pre-checks, but don't install anything.
      --etcd-image string        The etcd image to install.
      --generate-custom-yaml     Generate YAML files, but don't install anything.
  -h, --help                     help for install
      --k8s-timeout duration     The number of seconds to wait before timing out on␣
↪Kubernetes operations. (default 3m0s)
      --pv string                The name of the PV used by Trident.
      --pvc string               The name of the PVC used by Trident.
      --silent                   Disable most output during installation.
      --trident-image string     The Trident image to install.
      --use-custom-yaml          Use any existing YAML files that exist in setup␣
↪directory.
```

(continues on next page)

(continued from previous page)

```
      --volume-name string        The name of the storage volume used by Trident.
      --volume-size string        The size of the storage volume used by Trident.␣
↪(default "2Gi")
```

8.5 logs

Print the logs from Trident

```
Usage:
  tridentctl logs [flags]

Flags:
  -a, --archive      Create a support archive with all logs unless otherwise␣
↪specified.
  -h, --help         help for logs
  -l, --log string   Trident log to display. One of trident|etcd|auto|all (default
↪"auto")
  -p, --previous     Get the logs for the previous container instance if it exists.
```

8.6 uninstall

Uninstall Trident

```
Usage:
  tridentctl uninstall [flags]

Flags:
  -a, --all                       Deletes almost all artifacts of Trident, including␣
↪the PVC and PV used by Trident;
                                  however, it doesn't delete the volume used by␣
↪Trident from the storage backend. Use with caution!
  -h, --help                      help for uninstall
      --silent                    Disable most output during uninstallation.
```

8.7 update

Modify a resource in Trident

```
Usage:
  tridentctl update [command]

Available Commands:
  backend     Update a backend in Trident
```

8.8 version

Print the version of tridentctl and the running Trident service

```
Usage:
  tridentctl version
```

CHAPTER 9

REST API

While *tridentctl* is the easiest way to interact with Trident's REST API, you can use the REST endpoint directly if you prefer.

This is particularly useful for advanced installations that are using Trident as a standalone binary in non-Kubernetes deployments.

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

The API works as follows:

- `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 <trident-address>/trident/v1/<object-type>`: Creates an object of the specified type. Requires a JSON configuration for the object to be created; see the previous section for the specification of each object type. If the object already exists, behavior varies: backends update the existing object, while all other object types will fail the operation.
- `DELETE <trident-address>/trident/v1/<object-type>/<object-name>`: Deletes the named resource. Note that volumes associated with backends or storage classes will continue to exist; these must be deleted separately. See the section on backend deletion below.

To see an example of how these APIs are called, pass the debug (`-d`) flag to *tridentctl*.

CHAPTER 10

Simple Kubernetes install

Those that are interested in Trident and just getting started with Kubernetes frequently ask us for a simple way to install Kubernetes to try it out.

These instructions provide a bare-bones single node cluster that Trident will be able to integrate with for demonstration purposes.

> **Warning:** The Kubernetes cluster these instructions build should never be used in production. Follow production deployment guides provided by your distribution for that.

This is a simplification of the kubeadm install guide provided by Kubernetes. If you're having trouble, your best bet is to revert to that guide.

10.1 Prerequisites

An Ubuntu 16.04 machine with at least 1 GB of RAM.

These instructions are very opinionated by design, and will not work with anything else. For more generic instructions, you will need to run through the entire kubeadm install guide.

10.2 Install Docker CE 17.03

```
apt-get update && apt-get install -y curl apt-transport-https

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -
cat <<EOF >/etc/apt/sources.list.d/docker.list
deb https://download.docker.com/linux/$(lsb_release -si | tr '[:upper:]' '[:lower:]')
↪$(lsb_release -cs) stable
```

(continues on next page)

(continued from previous page)

```
EOF
apt-get update && apt-get install -y docker-ce=$(apt-cache madison docker-ce | grep␣
↪17.03 | head -1 | awk '{print $3}')
```

10.3 Install the appropriate version of kubeadm, kubectl and kubelet

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update
apt-get install -y kubeadm=1.13\* kubectl=1.13\* kubelet=1.13\* kubernetes-cni=0.6\*
```

10.4 Configure the host

```
swapoff -a
# Comment out swap line in fstab so that it remains disabled after reboot
vi /etc/fstab
```

10.5 Create the cluster

```
kubeadm init --kubernetes-version stable-1.13 --token-ttl 0 --pod-network-cidr=192.
↪168.0.0/16
```

10.6 Install the kubectl creds and untaint the cluster

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
kubectl taint nodes --all node-role.kubernetes.io/master-
```

10.7 Add an overlay network

```
kubectl apply -f https://docs.projectcalico.org/v2.6/getting-started/kubernetes/
↪installation/hosted/kubeadm/1.6/calico.yaml
```

10.8 Verify that all of the services started

After completing those steps, you should see output similar to this within a few minutes:

```
# kubectl get po -n kube-system
NAME                                        READY     STATUS    RESTARTS   AGE
calico-etcd-rvgzs                           1/1       Running   0          9d
calico-kube-controllers-6ff88bf6d4-db64s    1/1       Running   0          9d
calico-node-xpg2l                           2/2       Running   0          9d
etcd-scspa0333127001                        1/1       Running   0          9d
kube-apiserver-scspa0333127001              1/1       Running   0          9d
kube-controller-manager-scspa0333127001     1/1       Running   0          9d
kube-dns-545bc4bfd4-qgkrn                   3/3       Running   0          9d
kube-proxy-zvjcf                            1/1       Running   0          9d
kube-scheduler-scspa0333127001              1/1       Running   0          9d
```

Notice that all of the Kubernetes services are in a *Running* state. Congratulations! At this point your cluster is operational.

If this is the first time you're using Kubernetes, we highly recommend a walkthrough to familiarize yourself with the concepts and tools.