Trident Documentation

Release

NetApp

Apr 20, 2018

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

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 18.01.0:

```
wget https://github.com/NetApp/trident/releases/download/v18.01.0/trident-installer-
↪18.01.0.tar.gz
tar -xf trident-installer-18.01.0.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 *temporary* storage backend that the Trident installer will use *once* 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.

```
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

Run the Trident installer:

```
./install_trident.sh -n 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, Trident should be up and running in a few minutes.

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 |
+----------------+----------------+
| 18.01.0        | 18.01.0        |
+----------------+----------------+
```

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 you continue to see pods called `trident-launcher` or `trident-ephemeral` and a **Running** `trident-<generated id>` pod does not appear after a few minutes, Trident had a problem and the platform was *not* installed.

To help figure out what went wrong, you can view the logs by running:

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

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

```
./uninstall_trident.sh -n trident
```

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

2.1.6 5: Add your first backend

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

The installer does not assume that you want to use that backend configuration for the rest of the volumes that Trident provisions. So Trident forgot about it.

Create a storage backend configuration that Trident will provision volumes from. This can be the same backend configuration that you used in step 3, or something completely different. It's up to you.

```
./tridentctl -n trident create backend -f setup/backend.json
+-----------------------+----------------+--------+---------+
|         NAME          | STORAGE DRIVER | ONLINE | VOLUMES |
+-----------------------+----------------+--------+---------+
| ontapnas_10.0.0.1     | 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:

```
./uninstall_trident.sh -n <namespace>
./install_trident.sh -n <namespace>
```

By default the uninstall script 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.

Uninstalling Trident

The uninstall script in the installer bundle 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.

```
./uninstall_trident.sh -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` or `ontap-nas-economy` 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 service multipath-tools restart
```

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

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

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. You do have to explicitly set that option to use CHAP. Otherwise, see the *access groups guide* below.

Backend configuration options

Parameter	Description	Default
version	Always 1	
storageDriver-Name	Always "solidfire-san"	
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 (otherwise uses access groups)	false
AccessGroups	List of Access Group IDs to use	Finds the ID of an access group named "trident"
Types	QoS specifications (see below)	

Example configuration

```
{
    "version": 1,
    "storageDriverName": "solidfire-san",
    "Endpoint": "https://<user>:<password>@<mvip>/json-rpc/7.0",
    "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-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.

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 and ontap-nas-economy

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	
storageDriver-Name	"ontap-nas", "ontap-nas-economy" or "ontap-san"	
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"

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"
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": "netapp123",
    "defaults": {
      "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": "netapp123"
}
```

iSCSI Example for ontap-san driver

```
{
    "version": 1,
    "storageDriverName": "ontap-san",
    "managementLIF": "10.0.0.1",
    "dataLIF": "10.0.0.3",
    "svm": "svm_iscsi",
    "igroupName": "trident",
    "username": "vsadmin",
    "password": "netapp123"
}
```

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.

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"	
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"

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}]'
```

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 storageclass -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}]'
```

2.3 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-m048l     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
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/reclaimPolicy	N/A	any
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
trident.netapp.io/snapshotPolicy	snapshotPolicy	ontap-nas, ontap-nas-economy, ontap-san
trident.netapp.io/snapshotDirectory	snapshotDirectory	ontap-nas, ontap-nas-economy
trident.netapp.io/unixPermissions	unixPermissions	ontap-nas, ontap-nas-economy
trident.netapp.io/blockSize	blockSize	solidfire-san

The reclaim policy for the created PV can be determined by setting the annotation `trident.netapp.io/reclaimPolicy` in the PVC to either `Delete` or `Retain`; this value will then be set in the PV's `ReclaimPolicy` field. When the annotation is left unspecified, Trident will use the `Delete` policy. 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
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

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-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-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.

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`. 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
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` and `additionalStoragePools` 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. 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.

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.

2.6 Troubleshooting

- Run `tridentctl logs -l all -n trident`. Trident incorporates a multi-phase installation process that involves multiple pods, and the Trident pod itself includes multiple containers, all of which have their own logs. When something goes wrong, this is the best way to quickly examine all of the relevant logs.
- 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 here.

- If there's not enough information in the Trident and Trident launcher logs, you can try enabling debug mode for Trident and Trident launcher by passing the `-d` flag to the install script: `./install_trident.sh -d -n trident`.
- The *uninstall script* 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 service accounts are not available, the logs will report an error that `/var/run/secrets/kubernetes.io/serviceaccount/token` does not exist. In this case, you will either need to enable service accounts or connect to the API server by specifying the insecure address and port on the command line.

CHAPTER 3

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).

3.1 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",
       "username": "vsadmin",
   ```

```
    "password": "netapp123",
    "aggregate": "aggr1"
}
EOF
```

5. Start Trident using the managed plugin system.

```
docker plugin install netapp/trident-plugin:18.01 --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
```

3.2 Host and storage configuration

3.2.1 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/v18.01.0/trident-
↪installer-18.01.0.tar.gz
tar zxf trident-installer-18.01.0.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": "netapp123",
    "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
   ```

```
# 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": "netapp123",
    "aggregate": "aggr1"
}
EOF
```

3. Start Trident using the managed plugin system.

```
docker plugin install --grant-all-permissions --alias netapp netapp/trident-
↪plugin:18.01 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
```

3.2.2 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-san`, `eseries-iscsi`, or `solidfire-san`	ontap-nas
`storagePrefix`	Optional prefix for volume names. Default: "netappdvp_"	netappdvp_

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!

3.2.3 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	netapp123
aggregate	Aggregate to use for provisioning; it must be assigned to the SVM	aggr1

For the ontap-nas and ontap-nas-economy drivers a fully-qualified domain name can be specified for the 'dataLIF' option.

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

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

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
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.

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",
    "dataLIF": "10.0.0.2",
    "svm": "svm_nfs",
    "username": "vsadmin",
    "password": "netapp123",
    "aggregate": "aggr1",
    "defaults": {
      "size": "10G",
      "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": "netapp123",
    "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": "netapp123",
    "aggregate": "aggr1"
}
```

3.2.4 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/7.0",
    "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
            }
        }
    ]
}
```

3.2.5 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)	DockerHosts

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.

3.2.6 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:18.01 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:18.01 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
```

3.3 Upgrading

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:18.01
```

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
```

3.4 Common tasks

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.

3.4.1 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.
- `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 two 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

# 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
            }
        }
    ]
}
```

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
```

3.4.2 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, . . .).

3.4.3 Destroy a Volume

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

3.4.4 Volume Cloning

When using the ontap-nas, ontap-san, and solidfire-san storage drivers, the Docker Volume Plugin can clone volumes.

```
# 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
```

3.4.5 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).

3.5 Known issues

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

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

2. Because Docker Swarm does not orchestrate volume creation across multiple nodes, only the ontap-nas and ontap-san drivers will work in Swarm.

3.6 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.

3.6.1 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>
```

3.6.2 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 4

Requirements

4.1 Supported frontends (orchestrators)

Trident supports multiple container engines and orchestrators, including:

- Docker (CE and EE) 17.03, 17.06, 17.09, and 17.12
- Kubernetes 1.6, 1.7, 1.8, and 1.9
- OpenShift 3.6 and 3.7

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.

4.2 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 7 or later
- E/EF-Series SANtricity

4.3 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 and above

- Ubuntu 14.04 and above, 15.10 and above if using iSCSI multipathing
- CentOS 7.0 and above
- RHEL 7.0 and above
- CoreOS 1353.8.0 and above

The `tridentctl` utility also runs on any of these distributions of Linux.

4.4 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.

4.5 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.

4.6 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 5

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 6

trident

Trident exposes several command-line options. Normally the defaults will suffice, but you may want to modify them in your deployment. They are:

6.1 Logging

- -debug: Optional; enables debugging output.
- -loglevel <level>: Optional; sets the logging level (debug, info, warn, error, fatal). Defaults to info.

6.2 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.

6.3 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.

6.4 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.

6.5 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 7

tridentctl

The Trident installer bundle includes a command-line utility, `tridentctl`, that provides simple access to Trident. It can be used to interact with Trident 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
  logs        Print the logs from Trident
  version     Print the version of Trident

Flags:
  -d, --debug              Debug output
  -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
```

7.1 create

Add a resource to Trident

```
Usage:
  tridentctl create [command]

Available Commands:
  backend     Add a backend to Trident
```

7.2 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
```

7.3 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
```

7.4 logs

Print the logs from Trident

```
Usage:
  tridentctl logs [flags]

Flags:
  -a, --archive      Create a support archive with all logs unless otherwise␣
↪specified.
  -l, --log string   Trident log to display. One of␣
↪trident|etcd|launcher|ephemeral|auto|all (default "auto")
```

7.5 version

Print the version of tridentctl and the running Trident service

```
Usage:
  tridentctl version
```

CHAPTER 8

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 9

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.

9.1 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.

9.2 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
EOF
apt-get update && apt-get install -y docker-ce=$(apt-cache madison docker-ce | grep␣
↪17.03 | head -1 | awk '{print $3}')
```

9.3 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=1.8\* kubectl=1.8\* kubelet=1.8\*
```

9.4 Configure the host

```
swapoff -a
# Comment out swap line in fstab so that it remains disabled after reboot
vi /etc/fstab
```

9.5 Create the cluster

```
kubeadm init --kubernetes-version stable-1.8 --token-ttl 0 --pod-network-cidr=192.168.
↪0.0/16
```

9.6 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-
```

9.7 Add an overlay network

```
kubectl apply -f https://docs.projectcalico.org/v2.6/getting-started/kubernetes/
↪installation/hosted/kubeadm/1.6/calico.yaml
```

9.8 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.