Ansible Storage Role Documentation

Release 0.0.1alpha

Gorka Eguileor

Sep 13, 2019

Contents:

The Ansible Storage Role is a vendor agnostic abstraction providing infrastructure administrators with automation for storage solutions and to access provisioned resources.

CHAPTER 1

Introduction

The Ansible Storage Role is a vendor agnostic abstraction providing infrastructure administrators with automation for storage solutions and to access provisioned resources.

Thanks to this abstraction it's now possible to write reusable playbooks that can automate tasks on any of the supported storage arrays.

The role will provide an abstraction for multiple storage types:

- *Block* storage.
- *Shared* filesystems.
- *Object* storage.

Use cases:

- Automate provisioning of volumes for:
  - Bare metal hosts.
  - VMs managed via the virt Ansible module.
  - VMs managed on oVirt, OpenStack, and VMWare.
  - Cloud providers.
- Take periodical snapshots of provisioned volumes.

1.1 Features

At the moment the only supported storage type is *Block* storage, with a limited number of features:

- Get *backend* stats
- Create volumes
- Delete volumes

- Extend volumes
- Attach volumes
- Detach volumes

There are plans to add new features and provider for new storage types. Refer to the *Future work* section for information on the plans for the role.

1.2 Concepts

The Storage Role includes support for over 80 block storage drivers out of the box with the default provider, and this list can be expanded even further with new storage providers.

A provider is the Ansible module responsible for carrying out operations on the storage hardware. Each provider must support at least one specific hardware from a vendor, but it may as well support more, like the default provider does.

Even though there are only two providers at the moment, they support a large number of different storage vendors and storage backends.

To expose the functionality of these providers, the Storage Role introduces the concept of *backends*. A *backend* is constructed passing a specific configuration to a provider in order to manage a specific storage hardware.

There are two types of nodes in the Storage Role, *controllers* and *consumers*.

Fig. 1: Ansible Storage Role nodes diagram

Controllers have access to the storage management network and know how to connect to the storage hardware management interface and control it. For example to create and export a volume.

Consumers only need access to the storage data network in order to connect to the resources we have provisioned. For example to connect a volume via iSCSI.

1.3 Configuration

Before we can provision or use our storage, we need to setup the *controller* node, the one that will manage our storage.

There are two types of configuration options: One provides global configuration options for the provider, and the other provides the configuration required to access the storage's management interface.

In both cases the valid contents for these configuration parameters depend on the provider being used, as each provider has different options.

The names of the parameters are:

- *storage_backends* is a dictionary providing the configuration for all the *backends* we want the controller node to manage.
- *storage_$PROVIDER_config* and *storage_$PROVIDER_consumer_config* are the global provider configuration options to over-ride the defaults. Providers are expected to provide sensible defaults to avoid users having to change these.

All the information related to these configuration options is available on the *providers' section*, but here's an example of how to setup a node to manage an XtremIO array:

```
- hosts: storage_controller
  vars:
    storage_backends:
      xtremio:
        volume_driver: cinder.volume.drivers.dell_emc.xtremio.XtremIOISCSIDriver
        san_ip: w.x.y.z
        xtremio_cluster_name: CLUSTER-NAME
        san_login: admin
        san_password: nomoresecrets
  roles:
      - {role: storage, node_type: controller}
```

1.4 Example

Assuming our playbook has already been configured a backend on the controller node, for example like we did above, we can proceed to use this backend to provision and use the volumes like this:

```
- hosts: storage_consumers
  roles:
      - {role: storage, node_type: consumer}
  tasks:
      - name: Create volume
        storage:
            resource: volume
            state: present
            size: 1
        register: vol

      - name: Connect volume
        storage:
            resource: volume
            state: connected
        register: conn

      - debug:
          msg: "Volume {{ vol.id }} attached to {{ conn.path }}"

      - name: Disconnect volume
        storage:
            resource: volume
            state: disconnected

      - name: Delete volume
        storage:
            resource: volume
            state: absent
```

CHAPTER 2

Installation

Like with any other role, before you can use this role in your playbooks you'll need to install it in the system where the playbooks are going to be run:

```
$ ansible-galaxy install Akrog.storage
```

Once installed, you can use it in your playbooks. Usage of the role is covered in the *Usage* section.

The role has been tested with Ansible >= 2.4.1, but other versions may also work.

2.1 Requirements

Ansible Storage Role providers have specific requirements to manage and connect to the storage.

The Ansible Storage Role will try to automatically handle all the requirements for the nodes based on the selected provider and type of node. This means that using the storage role on nodes will install packages in order to perform the node's tasks (manage storage, or consume storage).

Attention: Right now requirements management has only been included for Fedora, CentOS, and RHEL.

Each storage provider has its own requirements, and they are usually different for the controller and the consumer nodes. Being lighter on the consumer nodes. Refer to the *providers section* for information on the requirements of each provider.

2.2 Consumer requirements

At the time of this writing the consumer role can't auto detect dependencies based on the connection type of the backends. Though we expect this to change in the future, at the moment any connection specific packages to connect volumes, need to be already installed in the system or added via tasks in the playbook.

Below are some of the packages required to use:

- *Multipathing*
- *iSCSI*
- *Ceph/RBD*

Other connection types will have different requirements. Please report an issue for any missing connection types and we'll add them.

2.2.1 Multipathing

Block storage multipathing requires package *device-mapper-multipath* to be installed, configured, and running. We can do this with a task or in the command line:

```
# yum install device-mapper-multipath
# mpathconf --enable --with_multipathd y \
> --user_friendly_names n \
> --find_multipaths y
# systemctl enable --now multipathd
```

Or as Ansible tasks:

```
- name: Install multipath package
  package:
    name: device-mapper-multipath
    state: present
 become: yes

- name: Create configuration
  command: mpathconf --enable --with_multipathd y --user_friendly_names n --find_
↪multipaths y
  args:
      creates: /etc/multipath.conf
 become: yes

- name: Start and enable on boot the multipath daemon
  service:
      name: multipathd
      state: started
      enabled: yes
 become: yes
```

2.2.2 iSCSI

To use iSCSI we need to install, configure, and run the *iscsi-initiator-utils* package if it's not already there:

```
# yum install iscsi-initiator-utils
# [ ! -e /etc/iscsi/initiatorname.iscsi ] \
> && echo InitiatorName=`iscsi-iname` > /etc/iscsi/initiatorname.iscsi
# systemctl enable --now iscsid
```

Or as Ansible tasks:

```
- name: Install iSCSI package
  package:
    name: iscsi-initiator-utils
    state: present
 become: yes

- name: Create initiator name
  shell: echo InitiatorName=`iscsi-iname` > /etc/iscsi/initiatorname.iscsi
  args:
      creates: /etc/iscsi/initiatorname.iscsi
 become: yes

- name: Start and enable on boot the iSCSI initiator
  service:
      name: iscsid
      state: started
      enabled: yes
 become: yes
```

2.2.3 Ceph/RBD

For Ceph/RBD connections we need to install the *ceph-common* package.

CHAPTER 3

Usage

In this section we'll cover how to use the storage role, the different operations available, their return values, how to address resources in the operations, and several examples.

One of the biggest differences between the Storage Role and other roles is that in this role it is recommended to include your storage tasks on the *consumer* nodes, even if part of the tasts are actually executed by the *controller*.

Instead of creating a task for the *controller* node to create as many volumes as *consumer* nodes we have and store the results in variables (or use a naming template), and then on the *consumer* nodes have a task that attaches one of those volumes to each node, we just have a task on the *consumers* to create the volume and connect it.

This way there's no need for variables or naming templates, and the creation and attaching tasks are together. This helps simplify the playbooks and the number of variables we have to move around in our playbooks, resulting in greater readability.

3.1 Configuration

The role needs to know what type of node we are defining, this is done using the *node_type* parameter. Acceptable values are *controller* and *consumer*. The default being *consumer*.

Note: When a node acts as controller and consumer we have to define it as two separate role entries. There is no *controller-consumer* or *all* node types.

Here's an example of how to configure a node to be the *controller* and a *consumer*.

```
- hosts: storage_controller
  vars:
      [ ... ]
  roles:
      - { role: storage, node_type: controller }
      - { role: storage, node_type: consumer }
```

For a *controller* node, the role needs to know the *backends* it's going to be managing in order to set them up. A single *controller* node can manage multiple *backends*, which are configured using the *storage_backends* variable.

The keys of the *storage_backends* dictionary define the IDs of the *backends* and must be preserved between runs to be able to access previously provisioned resources. If we change the *backend* IDs (key in the dictionary) we will no longer be able to access older resources.

The value part of each entry in the *storage_backends* dictionary corresponds to another dictionary, this one with the configuration of the specific *backend*. The key-value pairs in this dictionary will vary from one *provider* to another. The only shared key between them is the *provider* key used to select the provider we want to use for this backend.

The default value for the *provider* key is *cinderlib*, which is the default provider. When using the default value it is common practice to not include the *provider* key from the configuration.

We can have *backends* from different providers configured on the same *controller* node. For example, we can have one using the default provider and another using the *cinderclient* provider.

```
- hosts: storage_controller
  vars:
      storage_backends:
          backend1:
              [ ... ]
          backend2:
              provider: cinderclient
              [ ... ]
  roles:
      - { role: storage, node_type: controller }
```

A list of available parameters we can pass to each provider can be found in the *providers' section*.

Attention: *Controller* nodes must always be defined and setup in the playbooks before any storage can be used on a consumer node.

3.2 Resource addressing

In this section we'll cover the rules that are applied by the role to locate resources for the purposes of idempotency and resource addressing.

The storage role is modestly smart about locating resources, reducing the amount of information required to pass on task.

Volumes, which are the primary resource available at this moment, have the following attributes:

- *resource*: Type of the resource, must be *volume*.
- *backend*: Backend id.
- *provider*: Provider for the backend.
- *host*: Who "owns" this backend.
- *id*: UUID for the resource.
- *name*: User defined identifier for the volume.
- *size*: Size of the volume in GBi.

The way providers identify resources is by applying the parameters passed to tasks as if they were filters. If the result of applying the filters returns more than one resource, the provider will return an error.

For single *backend* controllers there's no need to pass *backend* or *provider* parameters, as they will default to the only configured *backend*. If we have configured multiple *backends* and at lest one of them is the default *provider*, then it will default to the first *backend* that was added. If there are multiple *backends* and none of them uses the default *provider*, then the role won't be able to determine a default value for these parameters.

Default value for *host* is the FQDN of the consumer node. Thanks to this, if we create resources as recommended, in a task on the consumer node, we won't need to create complicated templates to address volumes when performing tasks on multiple consumers.

Now that we know the basics of addressing resources it's probably best to have a look at examples of how it affects operations. In each one of the *Operations* we'll present different addressing situations using the *backends* defined in the previous *Configuration* section, where we have 2 *backends*:

- backend1 using the *cinderlib provider*.
- backend2 using the *cinderclient provider*.

3.3 Operations

3.3.1 Create

The most basic, and most common, operation is creating a volume on a *backend*, which is accomplished by setting the *state* of a *volume resource* to *present*. The default *state* for a *volume* is present, so there's no need to pass it. There are only 2 required attributes that must be passed on a create task: *resource* and *size*.

The task provides the following keys in the returned value at the root level:

Key	Contents
type	Type of resource. Now it can only be *volume*.
backend	ID of the backend where the volume exists. Matches the key provided in *storage_backends*.
host	Who "owns" this backend.
id	Resource's ID generated by the *provider*. Most providers use a UUID.
name	User defined identifier for the volume.
size	Size of the volume in GBi.

Here's the smallest task that can be used to create a volume:

```
- storage:
      resource: volume
      size: 1
```

We only have 2 backends, and only one of them uses the default *provider*, so following the addressing rules the volume will be created on backend1. This create task is equivalent to:

```
- storage:
      resource: volume
      state: present
      size: 1
      backend: backend1
      provider: cinderlib
```

If we wanted to create the volume on backend2, we would have to specify the *backend* or the *provider*. Passing the *provider* is also enough as there's only 1 *backend* for each *provider*:

```
- storage:
      resource: volume
      size: 1
      backend: backend2
```

The rest of the parameters will use defaults (*state: present*) or be detected automatically based on provided parameters (*provider: cinderclient*).

Creating these 2 volumes on the same node doesn't require any additional parameters as each one is going to different *backends*:

```
- storage:
      resource: volume
      size: 1

- storage:
      resource: volume
      size: 1
      backend: backend2
```

But if we try to do the same to create 2 volumes of the same size on the same *backend* like this:

```
- storage:
      resource: volume
      size: 1

- storage:
      resource: volume
      size: 1
```

We will end with only 1 volume, as the second call will be considered as a repeated call by the controller node. And since these are idempotent operations no new volume will be created.

To create multiple volumes of the same size on the same *backend* we need to use the *name* attribute. Providing it just in one of the tasks is enough, but we recommend passing it to both:

```
- storage:
      resource: volume
      size: 1
      name: first-volume

- storage:
      resource: volume
      size: 1
      name: second-volume
```

If each one of our volumes has a different size, then we don't need to provide a name, as one call cannot be mistaken for the other:

```
- storage:
      resource: volume
      size: 1

- storage:
      resource: volume
      size: 2
```

3.3.2 Delete

Deleting a specific volume is accomplished by setting the *state* of a *volume resource* to *absent*. And there are no required parameters for this call, but we can provide as many as we wan to narrow the volume we want to delete to a single one.

The delete task only returns the *changed* key to reflect whether the volume was present, and therefore was deleted, or if it wasn't present in the first place.

To reference a volume for deletion we usually use the same parameters that were used on the create task. If we didn't pass any parameters on create, passing none as well on delete will remove that volume:

```
- storage:
      resource: volume
      size: 1

- storage:
      resource: volume
      state: absent
```

Warning: There is no confirmation required to delete a volume, and once the volume is deleted it is usually impossible to recover its contents, so we recommend specifying as may parameters as possible on deletion tasks.

We don't need to provide the same parameters that we used on the create method as long as we provide enough information. We can use the return value from the create task to do the addressing:

```
- storage:
      resource: volume
      size: 1
      name: my_volume
      backend: backend2
  register: volume

- storage:
      resource: volume
      state: absent
      id: "{{volume.id}}"
      backend: "{{volume.backend}}"
```

Note: Keep in mind that there is no global database that stores all the resources IDs. So when using multiple *backends*, even if an ID uniquely identifies a resource in all your *backends*, the Storage Role has no way of knowing on which *backend* it is, so the task needs enough parameters to locate it. That's why in the example above we pass the *backend* parameter to the delete task.

When describin the create task we saw how we could create 2 volumes without a name because they had different sizes. If we wanted to remove those volumes we would have to provide the sizes on the delete task, otherwise the task would fail because there are 2 volumes that matches the addressing.

```
- storage:
      resource: volume
      size: 1

- storage:
```

(continues on next page)

(continued from previous page)

```
      resource: volume
      size: 2

- storage:
      resource: volume
      state: absent
      size: 1

- storage:
      resource: volume
      state: absent
      size: 2
```

3.3.3 Extend

Extending the size of a specific volume is accomplished by setting the *state* of a *volume resource* to *extended*. There is only one required parameters for this call, *size* which indicates the new size of the volume.

The task provides the following keys in the returned value at the root level:

Key Contents ========== == *changed* Following standard rules, will be *False* if the volume was

already connected, and *True* if it wasn't but now it is.

type **Describes the type of device that is connected, which at the** moment can only be *block*.

path Path to the device that has been added on the system. *additional_data* (Optional) *Provider* specific additional information. ========== ==

If we only have 1 volume on the node the addressing for the connect task is minimal.

```
- storage:
      resource: volume
      size: 1

- storage:
      resource: volume
      state: connected
```

Creating and connecting a volume is usually just the first step in our automation, and following tasks will rely on the *path* key of the returned value to use the volume on the *consumer* node.

```
- storage:
      resource: volume
      size: 1
  register: vol

- storage:
      resource: volume
      state: connected
  register: conn

- debug:
      msg: "Volume {{vol.id}} is now attached to {{conn.path}}"
```

3.3.4 Disconnect

Disconnecting a volume from a node is a multi-step process that undoes the steps performed during the connection in reverse. The *consumer* node detaches the volume from the node, and then the *controller* unmaps and removes the exported volume. These steps are opaque to the playbooks, where they are seen as a single task.

Disconnecting a specific volume from a node is accomplished by setting the *state* of a *volume resource* to *disconnected*. There are no specific parameters for the disconnect task. All parameters are used for the addressing of the volume. Addressing rules explained before apply here.

The disconnect task only returns the *changed* key to reflect whether the volume was present, and therefore was disconnected, or if it wasn't present in the first place.

Note: Disconnecting a volume will properly flush devices before proceeding to detach them. If it's a multipath device, the multipath will be flushed first and then the individual paths. If flushing is not possible due to connectivity issues the volume won't be disconnected.

When we using a single volume the disconnect doesn't need any additional parameters:

```
- storage:
      resource: volume
      size: 1

- storage:
      resource: volume
      state: connected

- storage:
      resource: volume
      state: disconnected
```

It's when we have multiple volumes that we have to provide more parameters, like we do in all the other tasks.

```
- storage:
      resource: volume
      size: 1

- storage:
      resource: volume
      size: 1
      backend: backend2

- storage:
      resource: volume
      backend: backend2
      state: connected

- storage:
      resource: volume
      backend: backend2
      state: disconnected
```

3.3.5 Stats

This is the only task that is meant to be executed on the *controller* node.

Stats gathering is a *provider* specific task that return arbitrary data. Each provider specifies what information is returned in the *providers' section*, but they must all return this data as the value for the *result* key.

And example for the default provider:

```
- storage:
      resource: backend
      backend: lvm
      state: stats
  register: stats

- debug:
      msg: "Backend {{stats.result.volume_backend_name}} from vendor {{stats.result.
↪vendor_name}} uses protocol {{stats.result.storage_protocol}}"
```

CHAPTER 4

Examples

On the *Introduction* and *Usage* sections we provided some examples and snippets. Here we'll provide larger examples to show the specifics of some backends, some interesting concepts, and advanced usage:

4.1 Kaminario backend

In this example we'll see how to configure the Kaminario K2 backend on a *controller* node using the default *cinderlib* provider.

Note: The Kaminario backend requires the *krest* PyPi package to be installed on the *controller*, but we don't need to worry about it, because the *cinderlib* provider takes care of it during the role setup.

```
- hosts: storage_controller
  vars:
    storage_backends:
        kaminario:
            volume_driver: cinder.volume.drivers.kaminario.kaminario_iscsi.
↪KaminarioISCSIDriver
              san_ip: w.x.y.z
              san_login: admin
              san_password: nomoresecrets
  roles:
      - { role: storage, node_type: controller }
```

4.2 Faster playbooks

If we are already certain that a node has everything installed, we can skip the setup part using variables *storage_setup_providers* and *storage_setup_consumers*.

For the controller:

```
- hosts: storage_controller
  vars:
    ansible_become: yes
    storage_setup_providers: no
    storage_backends:
        lvm:
            volume_driver: 'cinder.volume.drivers.lvm.LVMVolumeDriver'
            volume_group: 'ansible-volumes'
            target_protocol: 'iscsi'
            target_helper: 'lioadm'
  roles:
      - {role: storage, node_type: controller}
```

And for the consumers:

```
- hosts: storage_consumers
  vars:
    ansible_become: yes
    storage_setup_consumers: no
  roles:
      - {role: storage, node_type: consumer}
```

4.3 Populating data

Some applications may require specific data to be present in the system before they are run.

Thanks to the Storage Role we can easily automate the deployment of our whole application with custom configuration in an external disk:

- Install the application.
- Create a volume.
- Connect the volume.
- Format the volume.
- Populate the default configuration and data.
- Enable and start our application.

```
- hosts: storage_consumers
  roles:
      - {role: storage, node_type: consumer}
  tasks:
      - name: Install our application
        package:
            name: my-app
            state: present

      - name: Create the volume
        storage:
            resource: volume
            size: 20

      - name: Connect the volume
        storage:
```

(continues on next page)

(continued from previous page)

```
            resource: volume
            state: connected
        register: conn

      - name: Format the disk
        filesystem:
            fstype: ext4
            dev: "{{conn.path}}"
        become: yes

      - name: Mount the disk
        mount:
            path: /mnt/my-app-data
            src: "{{conn.path}}"
            fstype: ext4
            mode: 0777
        become: yes

      - name: Get default configuration and data
        unarchive:
            remote_src: yes
            src: https://mydomain.com/initial-data.tar.gz
            dest: /mnt/my-app-data
            owner: myapp
            group: myapp
        creates: /mnt/my-app-data/lib

      - name: Link the data to the disk contents
        file:
            src: /mnt/my-app-data/lib
            dest: /var/lib/my-app
            owner: myapp
            group: myapp
            state: link

      - name: Link the configuration to the disk contents
        file:
            src: /mnt/my-app-data/etc
            dest: /etc/my-app
            owner: myapp
            group: myapp
            state: link

      - name: Enable and start the service
        service:
            enabled: yes
            name: my-app
            state: started
```

4.4 Ceph backend

Unlike other *backends*, the Ceph/RBD backend does not receive all the *backend* configuration and credentials via parameters. It needs 2 configuration files present on the *controller* node, and the parameters must point to these files. The role doesn't know if these configuration files are already present on the *controller* node, if they must be copied

from the Ansible controller, or from some other locations, so it's our responsibility to copy them to the *controller* node.

Note: The Ceph/RBD backend requires the *ceph-common* package to be installed on the *controller*, but we don't need to worry about it, because the *cinderlib* provider takes care of it during the role setup.

Contents of our *ceph.conf* file:

```
[global]
fsid = fb86a5b7-6473-492d-865c-60229c986b8a
mon_initial_members = localhost.localdomain
mon_host = 192.168.1.22
auth_cluster_required = cephx
auth_service_required = cephx
auth_client_required = cephx
filestore_xattr_use_omap = true
osd crush chooseleaf type = 0
osd journal size = 100
osd pool default size = 1
rbd default features = 1
```

Contents of our *ceph.client.cinder.keyring* file:

```
[client.cinder]
        key = AQAj7eZarZzUBBAAB72Q6CjCqoftz8ISlk5XKg==
```

Here's how we would setup our *controller* using these files:

```
- hosts: storage_controller
  tasks:
      - file:
            path=/etc/ceph/
            state=directory
            mode: 0755
        become: yes
      - copy:
            src: ceph.conf
            dest: /etc/ceph/ceph.conf
            mode: 0644
        become: yes
      - copy:
            src: ceph.client.cinder.keyring
            dest: /etc/ceph/ceph.client.cinder.keyring
            mode: 0600
            owner: vagrant
            group: vagrant
        become: yes

- hosts: storage_controller
  vars:
    storage_backends:
        ceph:
            volume_driver: cinder.volume.drivers.rbd.RBDDriver
            rbd_user: cinder
            rbd_pool: volumes
            rbd_ceph_conf: /etc/ceph/ceph.conf
```

(continues on next page)

(continued from previous page)

```
            rbd_keyring_conf: /etc/ceph/ceph.client.cinder.keyring
  roles:
      - {role: storage, node_type: controller}
```

Note: The storage role runs a minimum check on the *backend* during setup, so we need to have the configuration files present before setting up the role.

By default, the RBD client looks for the keyring under */etc/ceph/* regardless of the configuration of the *rbd_keyring_conf* for the *backend*. If we want to have the keyring in another location we need to point it in the *cinder.conf* file.

Here's an example of how to store the keyring file out of the */etc/ceph* directory.

```
- hosts: storage_controller
  tasks:
      - file:
            path=/home/vagrant/ceph
            state=directory
            owner=vagrant
            group=vagrant
      - copy:
            src: ceph.conf
            dest: /home/vagrant/ceph/ceph.conf
      - copy:
            src: ceph.client.cinder.keyring
            dest: /home/vagrant/ceph/ceph.client.cinder.keyring
      - ini_file:
            dest=/home/vagrant/ceph/ceph.conf
            section=global
            option=keyring
            value=/home/vagrant/ceph/$cluster.$name.keyring

- hosts: storage_controller
  vars:
    storage_backends:
        ceph:
            volume_driver: cinder.volume.drivers.rbd.RBDDriver
            rbd_user: cinder
            rbd_pool: volumes
            rbd_ceph_conf: /home/vagrant/ceph/ceph.conf
            rbd_keyring_conf: /home/vagrant/ceph/ceph.client.cinder.keyring
  roles:
      - {role: storage, node_type: controller}
```

Attention: Even if we are setting they *keyring* in the *ceph.conf* file we must always pass the right *rbd_keyring_conf* parameter or we won't be able to attach from non controller nodes.

4.5 Bulk create

One case were we would be running a creation task on the controller would be if we want to have a pool of volumes at our disposal.

In this case we'll want to keep the *host* empty so it doesn't get the *controller* node's FQDN.

Here's an example creating 50 volumes of different sizes:

```
- hosts: storage_controller
  vars:
      num_disks: 50
      storage_backends:
          lvm:
              volume_driver: 'cinder.volume.drivers.lvm.LVMVolumeDriver'
              volume_group: 'cinder-volumes'
              target_protocol: 'iscsi'
              target_helper: 'lioadm'
  roles:
      - {role: storage, node_type: controller}
  tasks:
      - name: "Create {{num_disks}} volumes"
        storage:
            resource: volume
            state: present
            name: "mydisk{{item}}"
            host: ''
            size: "{{item}}"
        with_sequence: start=1 end={{num_disks}}
```

When using this kind of volumes we have to be careful with the addressing, because an undefined *host* parameter will default to the node's FQDN, which won't match the created volumes.

We can use the *name* parameter to connect to a volume, or we can use the size, if they are all of different sizes.

```
- hosts: web_server
  roles:
      - {role: storage, node_type: consumer}
  tasks:
      - storage:
            resource: volume
            state: connected
            host: ''
            size: 20
        register: conn
```

4.6 Migrating data

There may come a time when we want to migrate a volume from one *backend* to another. For example when moving volumes from a local testing *backend* to a real backend.

There are at least two ways of doing it, copying the whole device, or mounting the system and synchronizing the contents.

For simplicity we'll only cover the easy case of copying the whole device, which works fine when the destination is a thick volume. If the destination is a thin volume we would be wasting space.

```
- hosts: storage_controller
  vars:
      storage_backends:
          lvm:
```

(continues on next page)

(continued from previous page)

```
            volume_driver: 'cinder.volume.drivers.lvm.LVMVolumeDriver'
            volume_group: 'cinder-volumes'
            target_protocol: 'iscsi'
            target_helper: 'lioadm'
          kaminario:
            volume_driver: cinder.volume.drivers.kaminario.kaminario_iscsi.
↪KaminarioISCSIDriver
            san_ip: w.x.y.z
            san_login: admin
            san_password: nomoresecrets
  roles:
      - {role: storage, node_type: controller}

- hosts: storage_consumer
  tasks:
      - name: Retrieve the existing volume information
        storage:
            resource: volume
            backend: lvm
            state: present
            name: data-disk
        register: vol

      - name: Create a new volume on the destination backend using the source␣
↪information.
        storage:
            resource: volume
            backend: kaminario
            state: present
            name: "{{vol.name}}"
            size: "{{vol.size}}"
            host: "{{vol.host}}"
        register: new_vol

      - storage:
            resource: volume
            backend: lvm
            state: connected
            id: "{{vol.id}}"
        register: conn

      - storage:
            resource: volume
            backend: kaminario
            state: connected
            id: "{{new_vol.id}}"
        register: new_conn

      - name: Copy the data
        command: "dd if={{conn.path}} of={{new_conn.path}} bs=1M"
        become: true

      - storage:
            resource: volume
            backend: lvm
            state: disconnected
            id: "{{vol.id}}"
```

(continues on next page)

(continued from previous page)

```
    - storage:
          resource: volume
          backend: kaminario
          state: disconnected
          id: "{{new_vol.id}}"
```

CHAPTER 5

Supported storage

Supported backends are separated by type of storage they provide:

- *Block devices*
- *Shared filesystems*
- *Object storage*

5.1 Block devices

Currently both Block storage *providers* (*cinderlib* and *cinderclient*) support the same storage solutions, as they both use the same driver code. The biggest difference in terms of backend support is that the *cinderclient* provider relies on a *Cinder* service deployment, and that's how all the drivers have been validated by the automated testing system. The *cinderlib* provider relies on the *cinderlib* library, which is still in the process of automating the testing, and for the time being has only been manually validated with a limited number of backends.

Unless stated otherwise, drivers have not been validated with *cinderlib*, so even though they should work, they may not.

List of supported drivers in alphabetical order:

- Blockbridge EPS
- Ceph/RBD[2]
- Coho Data NFS[1]
- Dell EMC PS
- Dell EMC ScaleIO
- Dell EMC Unity
- Dell EMC VMAX FC

[2] This driver has been validated with *cinderlib* as stated in its documentation
[1] NFS backends that use an image to provide block storage are not supported yet.

- Dell EMC VMAX iSCSI[2]
- Dell EMC VNX
- Dell EMC XtremIO FC[2]
- Dell EMC XtremIO iSCSI[2]
- Dell Storage Center FC
- Dell Storage Center iSCSI
- DISCO
- DotHill FC
- DotHill iSCSI
- DRBD
- EMC CoprHD FC
- EMC CoprHD iSCSI
- EMC CoprHD ScaleIO
- FalconStor FSS FC
- FalconStor FSS iSCSI
- Fujitsu ETERNUS DX S3 FC
- Fujitsu ETERNUS DX S3 iSCSI
- Generic NFS[1]
- HGST
- Hitachi HBSD iSCSI
- Hitachi Hitachi NFS[1]
- Hitachi VSP FC
- Hitachi VSP iSCSI
- HPE 3PAR FC
- HPE 3PAR iSCSI
- HPE LeftHand iSCSI
- HPE MSA FC
- HPE MSA iSCSI
- HPE Nimble FC
- HPE Nimble iSCSI
- Huawei FusionStorage
- Huawei OceanStor FC
- Huawei OceanStor iSCSI
- IBM DS8000
- IBM FlashSystem A9000
- IBM FlashSystem A9000R

- IBM FlashSystem FC
- IBM FlashSystem iSCSI
- IBM GPFS
- IBM GPFS NFS[1]
- IBM GPFS Remote
- IBM Spectrum Accelerate
- IBM Storwize V7000 FC
- IBM Storwize V7000 iSCSI
- IBM SVC FC
- IBM SVC iSCSI
- IBM XIV
- INFINIDAT InfiniBox
- Infortrend Eonstor DS FC
- Infortrend Eonstor DS iSCSI
- Kaminario K2
- Lenovo FC
- Lenovo iSCSI
- LVM[2]
- NEC M-Series FC
- NEC M-Series iSCSI
- NetApp 7-mode FC
- NetApp 7-mode iSCSI
- NetApp 7-mode NFS[1]
- NetApp C-mode FC
- NetApp C-mode iSCSI
- NetApp Data ONTAP NFS[1]
- NetApp E-Series FC
- NetApp E-Series iSCSI
- NexentaEdge iSCSI
- NexentaEdge NFS[1]
- NexentaStor iSCSI
- NexentaStor NFS[1]
- Oracle ZFSSA iSCSI
- Oracle ZFSSA NFS[1]
- ProphetStor FC
- ProphetStor iSCSI

- Pure FC
- Pure iSCSI
- QNAP iSCSI
- Quobyte USP
- Reduxio
- Sheepdog
- SolidFire[2]
- Synology iSCSI
- Tegile FC
- Tegile iSCSI
- Tintri
- Veritas Clustered NFS[1]
- Veritas HyperScale
- Violin V7000 FC
- Violin V7000 iSCSI
- Virtuozzo
- VMware vCenter
- Windows Smbfs
- X-IO ISE FC
- X-IO ISE iSCSI
- XTE iSCSI
- Zadara VPSA iSCSI/iSER

5.2 Shared filesystems

The Storage role has no Shared filesystem provider, so it doesn't support any backend at the moment.

5.3 Object storage

The Storage role has no Object storage provider, so it doesn't support any backend at the moment.

CHAPTER 6

Storage Providers

Providers are separated by type of storage they provide:

- *Block storage*
- *Shared filesystems*
- *Object storage*

6.1 Block storage

The Storage Role currently has 2 block storage *providers*:

- *Cinderlib*
- *Cinderclient*

Both use the same storage drivers, supporting the same storage solutions, but using different approaches. The *Supported storage* section provides a detailed list of supported backends.

The default provider is *cinderlib*, as it doesn't rely on any existing service.

6.1.1 Cinderlib

The *cinderlib* Storage *provider* uses the *cinderlib* Python library to leverage existing *Cinder* drivers outside of *OpenStack*, without running any of the *Cinder* services: API, Scheduler, and Volume.

And when we say that *cinderlib* uses the same drivers as *Cinder*, we don't mean that these drivers have been copied out of the *Cinder* repository. We mean that we install the same *openstack-cinder* package used by the *Cinder* services, and use the exact same driver code on our *controller* nodes.

Thanks to the *Cinder* package, this *provider* supports a *considerable number of different drivers*. Most of the storage drivers included in the package don't have external dependencies and can run as they are. But there is a small number of drivers that require extra packages or libraries to manage the storage.

The *cinderlib provider* has the mechanism to automatically install these packages when deploying a *controller* based on the *backend* configuration. At this moment the drivers supporting this automatic installation is not complete, though it is growing.

As we mentioned, the *provider* uses the *openstack-cinder* package, which has its advantages, but comes with the drawback of requiring more dependencies than needed, such as the messaging and service libraries.

This, together with the specific driver requirements that we may be using, make the *cinderlib* provider somewhat heavy in terms of packages being installed. Making the most common deployment model to have only one *controller* node for all the consumers. One way to do it is using the node running the Ansible engine as the controller.

There is only 1 fixed parameter that the *cinderlib provider* requires:

Key	Contents
volume_driver	Namespace of the driver.

All other parameters depend on the driver we are using, and we recommend looking into the specific driver configuration page for more information on what these parameters are. If the driver has been validated for the *cinderlib* library we can see which parameters where used in its documentation.

Here is an example for XtremIO storage:

```
- hosts: storage_controller
  vars:
    storage_backends:
        xtremio:
            volume_driver: cinder.volume.drivers.dell_emc.xtremio.XtremIOISCSIDriver
            san_ip: w.x.y.z
            xtremio_cluster_name: CLUSTER-NAME
            san_login: admin
            san_password: nomoresecrets
  roles:
      - {role: storage, node_type: controller}
```

When working with the *cinderlib* provider there's one thing we must be aware of, the metadata persistence.

Cinder drivers are not required to be stateless, so most of them store metadata in the *Cinder* database to reduce the number of queries to the storage backend.

Since we use the *Cinder* drivers as they are, we cannot be stateless either. We'll use the metadata persistence plugin mechanism to store the driver's information. At this moment there's only one plugin available, the database one, allowing us to store the metadata in many different database engines.

Attention: If the metadata is lost, then the *cinderlib* role will no longer be able to use any of the resource it has created.

Proper care is recommended when deciding where to store the metadata. It can be stored in an external database, in a replicated shared filesystem, etc.

The default configuration is to store it in a SQLite file called *storage_cinderlib.sqlite* in the SSH user's home directory:

```
storage_cinderlib_persistence:
    storage: db
    connection: sqlite:///storage_cinderlib.sqlite
```

But we can change it to use other databases passing the connection information using SQLAlchemy database URLs format in the *connection* key.

For example we could use a MySQL database:

```
- hosts: storage_controller
  vars:
    storage_cinderlib_persistence:
        storage: db
        connection: mysql+pymysql://root:stackdb@127.0.0.1/cinder?charset=utf8
```

In the future there will be more metadata persistence plugins, and they will be referenced in *cinderlib*'s metadata persistence plugins documentation.

Having covered the *controller* nodes, we'll now look into the *consumer* nodes.

The *consumer* code is executed on a *consumer* node when we want to connect or disconnect a volume to the node. To achieve this it implements 3 functions:

- Connect volume.
- Disconnect volume.
- Get connector information for the node.

Please have a look at the *Consumer requirements* section for relevant information on the dependencies for connections on the *consumer* node.

Connection and disconnections are mostly managed using the OS-Brick. Although there are some exceptions like for Ceph/RBD connections where we manage them ourselves.

To speed things when we receive a call to connect a volume that's already connected, we use a simple SQLite database. This may change in the future.

This database is stored by default on the SSH user's home using filename *storage_cinderlib_consumer.sqlite*. But we can change the location with the *storage_cinderlib_consumer_defaults* variable. Default configuration is:

```
storage_cinderlib_consumer_defaults:
  db_file: storage_cinderlib_consumer.sqlite
```

Note: In future releases the use of the SQLite database on the *consumer* may be removed.

6.1.2 Cinderclient

The *cinderclient* Storage *provider* wraps an *OpenStack Cinder* service to expose it in Ansible using the Storage Role abstraction.

Communication between the Storage *provider* and the *Cinder* service is accomplished via *Cinder*'s well defined REST API.

Relying on an external *Cinder* service to manage our block storage greatly reduces the dependencies required by the *controller* nodes. The only dependency is the *python2-cinderclient* package, making *controllers* for the *cinder provider* very light.

With this *provider*, deploying all our nodes as *controller* and *consumer* makes sense.

The *cinderclient provider* needs the following configuration parameters to connect to a *Cinder* service:

Key	Contents
username	*OpenStack* user name.
password	Password for *OpenStack* user.
project_name	*OpenStack* project/tenant name.
region_name	*OpenStack* region name.
auth_url	URL for the authentication endpoint.
volume_type	*Cinder* volume type to use. When left undefined *provider* will use *Cinder*'s default volume type.

There are no global configuration options for the *cinderclient provider*, so values stored in the *storage_cinderclient_defaults* variable won't be used.

Note: Current implementation only supports *Cinder* services that use *Keystone* as the identity service. Standalone *Cinder* is not currently supported.

Here's a configuration example for the *cinderclient provider* showing how to use the default volume type from *Cinder*:

```
- hosts: storage_controller
  vars:
    storage_backends:
        default:
            provider: cinderclient
            password: nomoresecret
            auth_url: http://192.168.1.22/identity
            project_name: demo
            region_name: RegionOne
            username: admin
  roles:
      - {role: storage, node_type: controller}
```

Using a specific volume type is very easy, we just need to add the *volume_type* parameter:

```
- hosts: storage_controller
  vars:
    storage_backends:
        default:
            provider: cinderclient
            password: nomoresecret
            auth_url: http://192.168.1.22/identity
            project_name: demo
            region_name: RegionOne
            username: admin
            volume_type: ceph
  roles:
      - {role: storage, node_type: controller}
```

Since the *cinderclient* and *cinderlib providers* use the same storage driver code, the connection information to the storage obtained by the *controller* node follows the same format. Since the connection information is the same, both *providers* use the same *consumer* library code to present the storage on the *consumer* node. Please refer to the *Cinderlib* provider section for more information on this *consumer* module.

Note: Managed resources will be visible within *OpenStack*, and therefore can be managed using *Horizon* (the web interface), or the *cinderclient* command line. We don't recommend mixing management tools, so it'd be best to only manage Storage Role resources using Ansible. To help isolate our resources we recommend using a specific tenant for

the Storage Role.

6.2 Shared filesystems

There are no Shared filesystem providers at the moment.

6.3 Object storage

There are no Object storage providers at the moment.

CHAPTER 7

Internals

In this section we'll go over the Storage Role internals to explain the architecture, flows, and other implementation details.

This information should help debug issues on existing roles, and provide details on how to implement new roles.

Warning: This section is still in an early conceptualization phase, so it's not worth reading.

Todo: Do this whole section

Topics to cover:

- Installation tasks for the providers.
- Driver specific installation tasks for the *cinderlib provider*.
- How we send work to a *controller* when requested on the *consumer*.
- How we separate methods on the *controller* and *consumer* code.
- Data returned by the different method on the *controller* and *consumer*.
- How to create a new provider using *storage_base.py* classes.
- How a *provider* can reuse the *cinderlib consumer* code.
- Describe workarounds that have been implemented using callback and lookup plugins.
- Explain why the work was split between consumer and controller:
  - less requirements on consumer nodes
  - consumers don't need access to the management network
  - reuse consumer code/requirements
- Example of a workflow for attach or detach.

```
- hosts: storage_consumers
  roles:
      - {role: storage, node_type: consumer}
  tasks:
      - name: Create volume
        storage:
            resource: volume
            state: present
            size: 1
        register: vol

      - name: Connect volume
        storage:
            resource: volume
            state: connected
        register: conn

      - debug:
          msg: "Volume {{ vol.id }} attached to {{ conn.path }}"

      - name: Disconnect volume
        storage:
            resource: volume
            state: disconnected

      - name: Delete volume
        storage:
            resource: volume
            state: absent
```

This will create a volume for each consumer host and attach it to the node, then display the path where it has been connected before proceeding to disconnect and delete it.

A descriptive explanation of above playbook is:

- Initialize the controller node: Installs required libraries on the controller
- For each consumer node: - Install required libraries on the consumer node - Create a volume: Created on the controller and associated to consumer - Attach the volume created for that node:
  - Controller node maps the volume to the node (other nodes can't connect)
  - Consumer uses iSCSI initiator to attach the volume
  - Display where the volume has been attached
  - Detach the volume:
    - Consumer detaches the volume
    - Controller unmaps the volume

CHAPTER 8

Future work

The project being at the early development stages means that the current features serve mostly to demonstrate the power behind a common storage abstraction, but are somewhat limited.

There is work being done to add new features, and the next planned features are:

- Volume cloning.
- Snapshot management.
- Extend volume.
- Amazon's Elastic Block Storage (EBS).
- Manila provider for Shared filesystem.
- S3 provider for object storage.
- GCS provider for object storage.