Cinder Library Documentation

Release 5.2.1.dev1

Cinder Contributors

Dec 07, 2023

CONTENTS

The Cinder Library, also known as cinderlib, is a Python library that leverages the Cinder project to provide an object oriented abstraction around Cinders storage drivers to allow their usage directly without running any of the Cinder services or surrounding services, such as KeyStone, MySQL or RabbitMQ.

The library is intended for developers who only need the basic CRUD functionality of the drivers and dont care for all the additional features Cinder provides such as quotas, replication, multi-tenancy, migrations, retyping, scheduling, backups, authorization, authentication, REST API, etc.

The library was originally created as an external project, so it didnt have the broad range of backend testing Cinder does, and only a limited number of drivers were validated at the time. Drivers should work out of the box, and well keep a list of drivers that have added the cinderlib functional tests to the driver gates confirming they work and ensuring they will keep working.

CHAPTER

ONE

FEATURES

- Use a Cinder driver without running a DBMS, Message broker, or Cinder service.
- Using multiple simultaneous drivers on the same application.
- Basic operations support:
  - Create volume
  - Delete volume
  - Extend volume
  - Clone volume
  - Create snapshot
  - Delete snapshot
  - Create volume from snapshot
  - Connect volume
  - Disconnect volume
  - Local attach
  - Local detach
  - Validate connector
  - Extra Specs for specific backend functionality.
  - Backend QoS
  - Multi-pool support
- Metadata persistence plugins:
  - Stateless: Caller stores JSON serialization.
  - Database: Metadata is stored in a database: MySQL, PostgreSQL, SQLite
  - Custom plugin: Caller provides module to store Metadata and cinderlib calls it when necessary.

CHAPTER

TWO

EXAMPLE

The following code extract is a simple example to illustrate how cinderlib works. The code will use the LVM backend to create a volume, attach it to the local host via iSCSI, and finally snapshot it:

```
import cinderlib as cl

# Initialize the LVM driver
lvm = cl.Backend(volume_driver='cinder.volume.drivers.lvm.LVMVolumeDriver',
                 volume_group='cinder-volumes',
                 target_protocol='iscsi',
                 target_helper='lioadm',
                 volume_backend_name='lvm_iscsi')

# Create a 1GB volume
vol = lvm.create_volume(1, name='lvm-vol')

# Export, initialize, and do a local attach of the volume
attach = vol.attach()

print('Volume %s attached to %s' % (vol.id, attach.path))

# Snapshot it
snap = vol.create_snapshot('lvm-snap')
```

CHAPTER

THREE

TABLE OF CONTENTS

3.1 Installation

The Cinder Library is an interfacing library that doesnt have any storage driver code, so it expects Cinder drivers to be installed in the system to run properly.

We can use the latest stable release or the latest code from master branch.

3.1.1 Stable release

Drivers

For Red Hat distributions the recommendation is to use RPMs to install the Cinder drivers instead of using *pip*. If we dont have access to the Red Hat OpenStack Platform packages we can use the RDO community packages.

On CentOS, the Extras repository provides the RPM that enables the OpenStack repository. Extras is enabled by default on CentOS 7, so you can simply install the RPM to set up the OpenStack repository:

```
# yum install -y centos-release-openstack-rocky
# yum install -y openstack-cinder
```

On RHEL and Fedora, youll need to download and install the RDO repository RPM to set up the OpenStack repository:

```
# yum install -y https://www.rdoproject.org/repos/rdo-release.rpm
# yum install -y openstack-cinder
```

We can also install directly from source on the system or a virtual environment:

```
$ virtualenv venv
$ source venv/bin/activate
(venv) $ pip install git+git://github.com/openstack/cinder.git@stable/rocky
```

Library

To install Cinder Library well use PyPI, so well make sure to have the pip command available:

```
# yum install -y python-pip
# pip install cinderlib
```

This is the preferred method to install Cinder Library, as it will always install the most recent stable release.

If you dont have pip installed, this Python installation guide can guide you through the process.

3.1.2 Latest code

Drivers

If we dont have a packaged version or if we want to use a virtual environment we can install the drivers from source:

```
$ virtualenv cinder
$ source cinder/bin/activate
$ pip install git+git://github.com/openstack/cinder.git
```

Library

The sources for Cinder Library can be downloaded from the Github repo to use the latest version of the library.

You can either clone the public repository:

```
$ git clone git://github.com/akrog/cinderlib
```

Or download the tarball:

```
$ curl  -OL https://github.com/akrog/cinderlib/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ virtualenv cinder
$ python setup.py install
```

3.1.3 Dependencies

Cinderlib has less functionality than Cinder, which results in fewer required libraries.

When installing from PyPi or source, well get all the dependencies regardless of whether they are needed by *cinderlib* or not, since the Cinder Python package specifies all the dependencies. Installing from packages may result in fewer dependencies, but this will depend on the distribution package itself.

To increase loading speed, and reduce memory footprint and dependencies, *cinderlib* fakes all unnecessary packages at runtime if they have not already been loaded.

This can be convenient when creating containers, as one can remove unnecessary packages on the same layer *cinderlib* gets installed to get a smaller containers.

If our application uses any of the packages *cinderlib* fakes, we just have to import them before importing *cinderlib*. This way *cinderlib* will not fake them.

The list of top level packages unnecessary for *cinderlib* are:

- castellan
- cursive
- googleapiclient
- jsonschema
- keystoneauth1
- keystonemiddleware
- oauth2client
- os-win
- oslo.messaging
- oslo.middleware
- oslo.policy
- oslo.reports
- oslo.upgradecheck
- osprofiler
- paste
- pastedeploy
- pyparsing
- python-barbicanclient
- python-glanceclient
- python-novaclient
- python-swiftclient
- python-keystoneclient
- routes
- webob

3.2 Usage

Thanks to the fully Object Oriented abstraction, instead of a classic method invocation passing the resources to work on, *cinderlib* makes it easy to hit the ground running when managing storage resources.

Once the *Cinder* and *cinderlib* packages are installed we just have to import the library to start using it:

```
import cinderlib
```

Note: Installing the *Cinder* package does not require to start any of its services (volume, scheduler, api) or auxiliary services (KeyStone, MySQL, RabbitMQ, etc.).

Usage documentation is not too long, and it is recommended to read it all before using the library to be sure we have at least a high level view of the different aspects related to managing our storage with *cinderlib*.

Before going into too much detail there are some aspects we need to clarify to make sure our terminology is in sync and we understand where each piece fits.

In *cinderlib* we have *Backends*, that refer to a storage arrays specific connection configuration so it usually doesnt refer to the whole storage. With a backend well usually have access to the configured pool.

Resources managed by *cinderlib* are *Volumes* and *Snapshots*, and a *Volume* can be created from a *Backend*, another *Volume*, or from a *Snapshot*, and a *Snapshot* can only be created from a *Volume*.

Once we have a volume we can create *Connections* so it can be accessible from other hosts or we can do a local *Attachment* of the volume which will retrieve required local connection information of this host, create a *Connection* on the storage to this host, and then do the local *Attachment*.

Given that *Cinder* drivers are not stateless, *cinderlib* cannot be either. Thats why there is a metadata persistence plugin mechanism to provide different ways to store resource states. Currently we have memory and database plugins. Users can store the data wherever they want using the JSON serialization mechanism or with a custom metadata plugin.

Each of the different topics are treated in detail on their specific sections:

3.2.1 Initialization

The cinderlib itself doesnt require an initialization, as it tries to provide sensible settings, but in some cases we may want to modify these defaults to fit a specific desired behavior and the library provides a mechanism to support this.

Library initialization should be done before making any other library call, including *Backend* initialization and loading serialized data, if we try to do it after other calls the library will raise an *Exception*.

Provided *setup* method is *cinderlib.Backend.global_setup*, but for convenience the library provides a reference to this class method in *cinderlib.setup*

The method definition is as follows:

```
@classmethod
def global_setup(cls, file_locks_path=None, root_helper='sudo',
                 suppress_requests_ssl_warnings=True, disable_logs=True,
```

(continues on next page)

(continued from previous page)

```
                non_uuid_ids=False, output_all_backend_info=False,
                project_id=None, user_id=None, persistence_config=None,
                fail_on_missing_backend=True, host=None,
                **cinder_config_params):
```

The meaning of the librarys configuration options are:

file_locks_path

Cinder is a complex system that can support Active-Active deployments, and each driver and storage backend has different restrictions, so in order to facilitate mutual exclusion it provides 3 different types of locks depending on the scope the driver requires:

- Between threads of the same process.
- Between different processes on the same host.
- In all the OpenStack deployment.

Cinderlib doesnt currently support the third type of locks, but that should not be an inconvenience for most cinderlib usage.

Cinder uses file locks for the between process locking and cinderlib uses that same kind of locking for the third type of locks, which is also what Cinder uses when not deployed in an Active-Active fashion.

Parameter defaults to *None*, which will use the path indicated by the *state_path* configuration option. It defaults to the current directory.

root_helper

There are some operations in *Cinder* drivers that require *sudo* privileges, this could be because they are running Python code that requires it or because they are running a command with *sudo*.

Attaching and detaching operations with *cinderlib* will also require *sudo* privileges.

This configuration option allows us to define a custom root helper or disabling all *sudo* operations passing an empty string when we know we dont require them and we are running the process with a non passwordless *sudo* user.

Defaults to *sudo*.

suppress_requests_ssl_warnings

Controls the suppression of the *requests* library SSL certificate warnings.

Defaults to *True*.

non_uuid_ids

As mentioned in the *Volumes* section we can provide resource IDs manually at creation time, and some drivers even support non UUID identificators, but since thats not a given validation will reject any non UUID value.

This configuration option allows us to disable the validation on the IDs, at the users risk.

Defaults to *False*.

output_all_backend_info

Whether to include the *Backend* configuration when serializing objects. Detailed information can be found in the *Serialization* section.

Defaults to *False*.

disable_logs

Cinder drivers are meant to be run within a full blown service, so they can be quite verbose in terms of logging, thats why *cinderlib* disables it by default.

Defaults to *True*.

project_id

Cinder is a multi-tenant service, and when resources are created they belong to a specific tenant/project. With this parameter we can define, using a string, an identifier for our project that will be assigned to the resources we create.

Defaults to *cinderlib*.

user_id

Within each project/tenant the *Cinder* project supports multiple users, so when it creates a resource a reference to the user that created it is stored in the resource. Using this this parameter we can define, using a string, an identifier for the user of cinderlib to be recorded in the resources.

Defaults to *cinderlib*.

persistence_config

Cinderlib operation requires data persistence, which is achieved with a metadata persistence plugin mechanism.

The project includes 2 types of plugins providing 3 different persistence solutions and more can be used via Python modules and passing custom plugins in this parameter.

Users of the *cinderlib* library must decide which plugin best fits their needs and pass the appropriate configuration in a dictionary as the *persistence_config* parameter.

The parameter is optional, and defaults to the *memory* plugin, but if its passed it must always include the *storage* key specifying the plugin to be used. All other key-value pairs must be valid parameters for the specific plugin.

Value for the *storage* key can be a string identifying a plugin registered using Python entrypoints, an instance of a class inheriting from *PersistenceDriverBase*, or a *PersistenceDriverBase* class.

Information regarding available plugins, their description and parameters, and different ways to initialize the persistence can be found in the *Metadata Persistence* section.

fail_on_missing_backend

To facilitate operations on resources, *Cinderlib* stores a reference to the instance of the *backend* in most of the in-memory objects.

When deserializing or retrieving objects from the metadata persistence storage *cinderlib* tries to properly set this *backend* instance based on the *backends* currently in memory.

Trying to load an object without having instantiated the *backend* will result in an error, unless we define *fail_on_missing_backend* to *False* on initialization.

This is useful if we are sharing the metadata persistence storage and we want to load a volume that is already connected to do just the attachment.

host

Host configuration option used for all volumes created by this cinderlib execution.

On cinderlib volumes are selected based on the backend name, not on the host@backend combination like cinder does. Therefore backend names must be unique across all cinderlib applications that are using the same persistence storage backend.

A second application running cinderlib with a different host value will have access to the same resources if it uses the same backend name.

Defaults to the hosts hostname.

Other keyword arguments

Any other keyword argument passed to the initialization method will be considered a *Cinder* configuration option in the *[DEFAULT]* section.

This can be useful to set additional logging configuration like debug log level, the *state_path* used by default in many option, or other options like the *ssh_hosts_key_file* required by drivers that use SSH.

For a list of the possible configuration options one should look into the *Cinder* projects documentation.

3.2.2 Backends

The *Backend* class provides the abstraction to access a storage array with an specific configuration, which usually constraint our ability to operate on the backend to a single pool.

Note: While some drivers have been manually validated most drivers have not, so theres a good chance that using any non tested driver will show unexpected behavior.

If you are testing *cinderlib* with a non verified backend you should use an exclusive pool for the validation so you dont have to be so careful when creating resources as you know that everything within that pool is related to *cinderlib* and can be deleted using the vendors management tool.

If you try the library with another storage array I would love to hear about your results, the library version, and configuration used (masked IPs, passwords, and users).

Initialization

Before we can have access to an storage array we have to initialize the *Backend*, which only has one defined parameter and all other parameters are not defined in the method prototype:

```
class Backend(object):
    def __init__(self, volume_backend_name, **driver_cfg):
```

There are two arguments that well always have to pass on the initialization, one is the *volume_backend_name* that is the unique identifier that *cinderlib* will use to identify this specific driver initialization, so well need to make sure not to repeat the name, and the other one is the *volume_driver* which refers to the Python namespace that points to the *Cinder* driver.

All other *Backend* configuration options are free-form keyword arguments. Each driver and storage array requires different information to operate, some require credentials to be passed as parameters, while others use a file, and some require the control address as well as the data addresses. This behavior is inherited from the *Cinder* project.

To find what configuration options are available and which ones are compulsory the best is going to the Vendors documentation or to the OpenStacks Cinder volume driver configuration documentation.

Cinderlib supports references in the configuration values using the forms:

- `$[<config_group>.]<config_option>`
- `${[<config_group>.]<config_option>}`

Where `config_group` is `backend_defaults` for the driver configuration options.

Attention: The `rbd_keyring_file` configuration parameter does not accept templating.

Examples:

- `target_ip_address='$my_ip'`
- `volume_group='my-${backend_defaults.volume_backend_name}-vg'`

Attention: Some drivers have external dependencies which we must satisfy before initializing the driver or it may fail either on the initialization or when running specific operations. For example Kaminario requires the *krest* Python library, and Pure requires *purestorage* Python library.

Python library dependencies are usually documented in the driver-requirements.txt file, as for the CLI required tools, well have to check in the Vendors documentation.

Cinder only supports using one driver at a time, as each process only handles one backend, but *cinderlib* has overcome this limitation and supports having multiple *Backends* simultaneously.

Lets see now initialization examples of some storage backends:

LVM

```
import cinderlib

lvm = cinderlib.Backend(
    volume_driver='cinder.volume.drivers.lvm.LVMVolumeDriver',
    volume_group='cinder-volumes',
    target_protocol='iscsi',
    target_helper='lioadm',
    volume_backend_name='lvm_iscsi',
)
```

XtremIO

```
import cinderlib

xtremio = cinderlib.Backend(
    volume_driver='cinder.volume.drivers.dell_emc.xtremio.XtremIOISCSIDriver',
    san_ip='10.10.10.1',
    xtremio_cluster_name='xtremio_cluster',
    san_login='xtremio_user',
    san_password='xtremio_password',
    volume_backend_name='xtremio',
)
```

Kaminario

```
import cinderlib

kaminario = cl.Backend(
    volume_driver='cinder.volume.drivers.kaminario.kaminario_iscsi.
↪KaminarioISCSIDriver',
    san_ip='10.10.10.2',
    san_login='kaminario_user',
```

(continues on next page)

(continued from previous page)

```
    san_password='kaminario_password',
    volume_backend_name='kaminario_iscsi',
)
```

For other backend configuration examples please refer to the *Validated drivers* page.

Available Backends

Usual procedure is to initialize a *Backend* and store it in a variable at the same time so we can use it to manage our storage backend, but there are cases where we may have lost the reference or we are in a place in our code where we dont have access to the original variable.

For these situations we can use *cinderlibs* tracking of *Backends* through the *backends* class dictionary where all created *Backends* are stored using the *volume_backend_name* as the key.

```
for backend in cinderlib.Backend.backends.values():
    initialized_msg = '' if backend.initialized else 'not '
    print('Backend %s is %sinitialized with configuration: %s' %
          (backend.id, initialized_msg, backend.config))
```

Installed Drivers

Available drivers for *cinderlib* depend on the Cinder version installed, so we have a method, called *list_supported_drivers* to list information about the drivers that are included with the Cinder release installed in the system.

The method accepts parameter `output_version` where we can specify the desired output format:

- `1` for human usage (default value).
- `2` for automation tools.

The main difference are the values of the driver options and how the expected type of these options is described.

```
import cinderlib

drivers = cinderlib.list_supported_drivers()
```

And what well get is a dictionary with the class name of the driver, a description, the version of the driver, etc.

Heres the entry for the LVM driver:

```
{'LVMVolumeDriver':
    {'ci_wiki_name': 'Cinder_Jenkins',
     'class_fqn': 'cinder.volume.drivers.lvm.LVMVolumeDriver',
     'class_name': 'LVMVolumeDriver',
     'desc': 'Executes commands relating to Volumes.',
     'supported': True,
     'version': '3.0.0',
```

(continues on next page)

(continued from previous page)

```
    'driver_options': [
        {'advanced': 'False',
         'default': '64',
         'deprecated_for_removal': 'False',
         'deprecated_opts': '[]',
         'deprecated_reason': 'None',
         'deprecated_since': 'None',
         'dest': 'spdk_max_queue_depth',
         'help': 'Queue depth for rdma transport.',
         'metavar': 'None',
         'mutable': 'False',
         'name': 'spdk_max_queue_depth',
         'positional': 'False',
         'required': 'False',
         'sample_default': 'None',
         'secret': 'False',
         'short': 'None',
         'type': 'Integer(min=1, max=128)'},
    ],
   }
},
```

The equivalent for the LVM driver for automation would be:

```
import cinderlib

drivers = cinderlib.list_supported_drivers(2)

 {'LVMVolumeDriver':
     {'ci_wiki_name': 'Cinder_Jenkins',
      'class_fqn': 'cinder.volume.drivers.lvm.LVMVolumeDriver',
      'class_name': 'LVMVolumeDriver',
      'desc': 'Executes commands relating to Volumes.',
      'supported': True,
      'version': '3.0.0',
      'driver_options': [
         {'advanced': False,
          'default': 64,
          'deprecated_for_removal': False,
          'deprecated_opts': [],
          'deprecated_reason': None,
          'deprecated_since': None,
          'dest': 'spdk_max_queue_depth',
          'help': 'Queue depth for rdma transport.',
          'metavar': None,
          'mutable': False,
          'name': 'spdk_max_queue_depth',
          'positional': False,
          'required': False,
          'sample_default': None,
```

(continues on next page)

(continued from previous page)

```
          'secret': False,
          'short': None,
          'type': {'choices': None,
                   'max': 128,
                   'min': 1,
                   'num_type': <class 'int'>,
                   'type_class': Integer(min=1, max=128),
                   'type_name': 'integer value'}}
      ],
     }
 },
```

Stats

In *Cinder* all cinder-volume services periodically report the stats of their backend to the cinder-scheduler services so they can do informed placing decisions on operations such as volume creation and volume migration.

Some of the keys provided in the stats dictionary include:

- *driver_version*
- *free_capacity_gb*
- *storage_protocol*
- *total_capacity_gb*
- *vendor_name volume_backend_name*

Additional information can be found in the Volume Stats section within the Developers Documentation.

Gathering stats is a costly operation for many storage backends, so by default the stats method will return cached values instead of collecting them again. If latest data is required parameter *refresh=True* should be passed in the *stats* method call.

Heres an example of the output from the LVM *Backend* with refresh:

```
>>> from pprint import pprint
>>> pprint(lvm.stats(refresh=True))
{'driver_version': '3.0.0',
 'pools': [{'QoS_support': False,
            'filter_function': None,
            'free_capacity_gb': 20.9,
            'goodness_function': None,
            'location_info': 'LVMVolumeDriver:router:cinder-volumes:thin:0',
            'max_over_subscription_ratio': 20.0,
            'multiattach': False,
            'pool_name': 'LVM',
            'provisioned_capacity_gb': 0.0,
            'reserved_percentage': 0,
            'thick_provisioning_support': False,
            'thin_provisioning_support': True,
```

(continues on next page)

(continued from previous page)

```
            'total_capacity_gb': '20.90',
            'total_volumes': 1}],
 'sparse_copy_volume': True,
 'storage_protocol': 'iSCSI',
 'vendor_name': 'Open Source',
 'volume_backend_name': 'LVM'}
```

Available volumes

The *Backend* class keeps track of all the *Backend* instances in the *backends* class attribute, and each *Backend* instance has a *volumes* property that will return a *list* all the existing volumes in the specific backend. Deleted volumes will no longer be present.

So assuming that we have an *lvm* variable holding an initialized *Backend* instance where we have created volumes we could list them with:

```
for vol in lvm.volumes:
    print('Volume %s has %s GB' % (vol.id, vol.size))
```

Attribute *volumes* is a lazy loadable property that will only update its value on the first access. More information about lazy loadable properties can be found in the *Resource tracking* section. For more information on data loading please refer to the *Metadata Persistence* section.

Note: The *volumes* property does not query the storage array for a list of existing volumes. It queries the metadata storage to see what volumes have been created using *cinderlib* and return this list. This means that we wont be able to manage pre-existing resources from the backend, and we wont notice when a resource is removed directly on the backend.

Attributes

The *Backend* class has no attributes of interest besides the *backends* mentioned above and the *id*, *config*, and JSON related properties well see later in the *Serialization* section.

The *id* property refers to the *volume_backend_name*, which is also the key used in the *backends* class attribute.

The *config* property will return a dictionary with only the volume backends name by default to limit unintended exposure of backend credentials on serialization. If we want it to return all the configuration options we need to pass *output_all_backend_info=True* on *cinderlib* initialization.

If we try to access any non-existent attribute in the *Backend*, *cinderlib* will understand we are trying to access a *Cinder* driver attribute and will try to retrieve it from the drivers instance. This is the case with the *initialized* property we accessed in the backends listing example.

Other methods

All other methods available in the *Backend* class will be explained in their relevant sections:

- *load* and *load_backend* will be explained together with *json*, *jsons*, *dump*, *dumps* properties and *to_dict* method in the *Serialization* section.
- *create_volume* method will be covered in the *Volumes* section.
- *validate_connector* will be explained in the *Connections* section.
- *global_setup* has been covered in the *Initialization* section.
- *pool_names* tuple with all the pools available in the driver. Non pool aware drivers will have only 1 pool and use the name of the backend as its name. Pool aware drivers may report multiple values, which can be passed to the *create_volume* method in the *pool_name* parameter.

3.2.3 Volumes

The *Volume* class provides the abstraction layer required to perform all operations on an existing volume. Volume creation operations are carried out at the *Backend* level.

Create

The base resource in storage is the volume, and to create one the *cinderlib* provides three different mechanisms, each one with a different method that will be called on the source of the new volume.

So we have:

- Empty volumes that have no resource source and will have to be created directly on the *Backend* via the *create_volume* method.
- Cloned volumes that will be created from a source *Volume* using its *clone* method.
- Volumes from a snapshot, where the creation is initiated by the *create_volume* method from the *Snapshot* instance.

Note: *Cinder* NFS backends will create an image and not a directory to store files, which falls in line with *Cinder* being a Block Storage provider and not filesystem provider like *Manila* is.

So assuming that we have an *lvm* variable holding an initialized *Backend* instance we could create a new 1GB volume quite easily:

```
print('Stats before creating the volume are:')
pprint(lvm.stats())
vol = lvm.create_volume(1)
print('Stats after creating the volume are:')
pprint(lvm.stats())
```

Now, if we have a volume that already contains data and we want to create a new volume that starts with the same contents we can use the source volume as the cloning source:

```
cloned_vol = vol.clone()
```

Some drivers support cloning to a bigger volume, so we could define the new size in the call and the driver would take care of extending the volume after cloning it, this is usually tightly linked to the *extend* operation support by the driver.

Cloning to a greater size would look like this:

```
new_size = vol.size + 1
cloned_bigger_volume = vol.clone(size=new_size)
```

Note: Cloning efficiency is directly linked to the storage backend in use, so it will not have the same performance in all backends. While some backends like the Ceph/RBD will be extremely efficient others may range from slow to being actually implemented as a *dd* operation performed by the driver attaching source and destination volumes.

```
vol = snap.create_volume()
```

Note: Just like with the cloning functionality, not all storage backends can efficiently handle creating a volume from a snapshot.

On volume creation we can pass additional parameters like a *name* or a *description*, but these will be irrelevant for the actual volume creation and will only be useful to us to easily identify our volumes or to store additional information.

Available fields with their types can be found in Cinders Volume OVO definition, but most of them are only relevant within the full *Cinder* service.

We can access these fields as if they were part of the *cinderlib Volume* instance, since the class will try to retrieve any non *cinderlib Volume* from *Cinder*s internal OVO representation.

Some of the fields we could be interested in are:

- *id*: UUID-4 unique identifier for the volume.
- *user_id*: String identifier, in *Cinder* its a UUID, but we can choose here.
- *project_id*: String identifier, in *Cinder* its a UUID, but we can choose here.
- *snapshot_id*: ID of the source snapshot used to create the volume. This will be filled by *cinderlib*.
- *host*: Used to store the backend name information together with the host name where cinderlib is running. This information is stored as a string in the form of *host@backend#pool*. This is an optional parameter, and passing it to *create_volume* will override default value, allowing our caller to request a specific pool for multi-pool backends, though we recommend using the *pool_name* parameter instead. Issues will arise if parameter doesnt contain correct information.
- *pool_name*: Pool name to use when creating the volume. Default is to use the first or only pool. To know possible values for a backend use the *pool_names* property on the *Backend* instance.
- *size*: Volume size in GBi.
- *availability_zone*: In case we want to define AZs.
- *status*: This represents the status of the volume, and the most important statuses are *available*, *error*, *deleted*, *in-use*, *creating*.

- *attach_status*: This can be *attached* or *detached*.
- *scheduled_at*: Date-time when the volume was scheduled to be created. Currently not being used by *cinderlib*.
- *launched_at*: Date-time when the volume creation was completed. Currently not being used by *cinderlib*.
- *deleted*: Boolean value indicating whether the volume has already been deleted. It will be filled by *cinderlib*.
- *terminated_at*: When the volume delete was sent to the backend.
- *deleted_at*: When the volume delete was completed.
- *display_name*: Name identifier, this is passed as *name* to all *cinderlib* volume creation methods.
- *display_description*: Long description of the volume, this is passed as *description* to all *cinderlib* volume creation methods.
- *source_volid*: ID of the source volume used to create this volume. This will be filled by *cinderlib*.
- *bootable*: Not relevant for *cinderlib*, but maybe useful for the *cinderlib* user.
- *extra_specs*: Extra volume configuration used by some drivers to specify additional information, such as compression, deduplication, etc. Key-Value pairs are driver specific.
- *qos_specs*: Backend QoS configuration. Dictionary with driver specific key-value pares that enforced by the backend.

Note: *Cinderlib* automatically generates a UUID for the *id* if one is not provided at volume creation time, but the caller can actually provide a specific *id*.

By default the *id* is limited to valid UUID and this is the only kind of ID that is guaranteed to work on all drivers. For drivers that support non UUID IDs we can instruct *cinderlib* to modify *Cinder*s behavior and allow them. This is done on *cinderlib* initialization time passing *non_uuid_ids=True*.

Note: *Cinderlib* does not do scheduling on driver pools, so setting the *extra_specs* for a volume on drivers that expect the scheduler to select a specific pool using them will have the same behavior as in Cinder.

In that case the caller of Cinderlib is expected to go through the stats and check the pool that matches the criteria and pass it to the Backends *create_volume* method on the *pool_name* parameter.

Delete

Once we have created a *Volume* we can use its *delete* method to permanently remove it from the storage backend.

In *Cinder* there are safeguards to prevent a delete operation from completing if it has snapshots (unless the delete request comes with the *cascade* option set to true), but here in *cinderlib* we dont, so its the callers responsibility to delete the snapshots.

Deleting a volume with snapshots doesnt have a defined behavior for *Cinder* drivers, since its never meant to happen, so some storage backends delete the snapshots, other leave them as they were, and others will fail the request.

Example of creating and deleting a volume:

```
vol = lvm.create_volume(size=1)
vol.delete()
```

Attention: When deleting a volume that was the source of a cloning operation some backends cannot delete them (since they have copy-on-write clones) and they just keep them as a silent volume that will be deleted when its snapshot and clones are deleted.

Extend

Many storage backends and *Cinder* drivers support extending a volume to have more space and you can do this via the *extend* method present in your *Volume* instance.

If the *Cinder* driver doesnt implement the extend operation it will raise a *NotImplementedError*.

The only parameter received by the *extend* method is the new size, and this must always be greater than the current value because *cinderlib* is not validating this at the moment.

The call will return the new size of the volume in bytes.

Example of creating, extending, and deleting a volume:

```
vol = lvm.create_volume(size=1)
print('Vol %s has %s GBi' % (vol.id, vol.size))
new_size = vol.extend(2)
print('Extended vol %s has %s GBi' % (vol.id, vol.size))
print('Detected new size is %s bytes' % new_size)
vol.delete()
```

A call to *extend* on a locally attached volume will automatically update the hosts view of the volume to reflect the new size. For non locally attached volumes please refer to the extend section in the connections section.

Other methods

All other methods available in the *Volume* class will be explained in their relevant sections:

- *load* will be explained together with *json*, *jsons*, *dump*, and *dumps* properties, and the *to_dict* method in the *Serialization* section.
- *refresh* will reload the volume from the metadata storage and reload any lazy loadable property that has already been loaded. Covered in the *Serialization* and *Resource tracking* sections.
- *create_snapshot* method will be covered in the *Snapshots* section together with the *snapshots* attribute.
- *attach*, *detach*, *connect*, and *disconnect* methods will be explained in the *Connections* section.

3.2.4 Snapshots

The *Snapshot* class provides the abstraction layer required to perform all operations on an existing snapshot, which means that the snapshot creation operation must be invoked from other class instance, since the new snapshot we want to create doesnt exist yet and we cannot use the *Snapshot* class to manage it.

Create

Once we have a *Volume* instance we are ready to create snapshots from it, and we can do it for attached as well as detached volumes.

Note: Some drivers, like the NFS, require assistance from the Compute service for attached volumes, so there is currently no way of doing this with *cinderlib*

Creating a snapshot can only be performed by the *create_snapshot* method from our *Volume* instance, and once we have created a snapshot it will be tracked in the *Volume* instances *snapshots* set.

Here is a simple code to create a snapshot and use the *snapshots* set to verify that both, the returned value by the call as well as the entry added to the *snapshots* attribute, reference the same object and that the *volume* attribute in the *Snapshot* is referencing the source volume.

```
vol = lvm.create_volume(size=1)
snap = vol.create_snapshot()
assert snap is list(vol.snapshots)[0]
assert vol is snap.volume
```

Delete

Once we have created a *Snapshot* we can use its *delete* method to permanently remove it from the storage backend.

Deleting a snapshot will remove its reference from the source *Volume*s *snapshots* set.

```
vol = lvm.create_volume(size=1)
snap = vol.create_snapshot()
assert 1 == len(vol.snapshots)
snap.delete()
assert 0 == len(vol.snapshots)
```

Other methods

All other methods available in the *Snapshot* class will be explained in their relevant sections:

- *load* will be explained together with *json*, *jsons*, *dump*, and *dumps* properties, and the *to_dict* method in the *Serialization* section.
- *refresh* will reload the volume from the metadata storage and reload any lazy loadable property that has already been loaded. Covered in the *Serialization* and *Resource tracking* sections.
- *create_volume* method has been covered in the *Volumes* section.

3.2.5 Connections

When talking about attaching a *Cinder* volume there are three steps that must happen before the volume is available in the host:

1. Retrieve connection information from the host where the volume is going to be attached. Here we would be getting iSCSI initiator name, IP, and similar information.
2. Use the connection information from step 1 and make the volume accessible to it in the storage backend returning the volume connection information. This step entails exporting the volume and initializing the connection.
3. Attaching the volume to the host using the data retrieved on step 2.

If we are running *cinderlib* and doing the attach in the same host, then all steps will be done in the same host. But in many cases you may want to manage the storage backend in one host and attach a volume in another. In such cases, steps 1 and 3 will happen in the host that needs the attach and step 2 on the node running *cinderlib*.

Projects in *OpenStack* use the *OS-Brick* library to manage the attaching and detaching processes. Same thing happens in *cinderlib*. The only difference is that there are some connection types that are handled by the hypervisors in *OpenStack*, so we need some alternative code in *cinderlib* to manage them.

Connection objects most interesting attributes are:

- *connected*: Boolean that reflects if the connection is complete.
- *volume*: The *Volume* to which this instance holds the connection information.
- *protocol*: String with the connection protocol for this volume, ie: *iscsi*, *rbd*.
- *connector_info*: Dictionary with the connection information from the host that is attaching. Such as its hostname, IP address, initiator name, etc.
- *conn_info*: Dictionary with the connection information the host requires to do the attachment, such as IP address, target name, credentials, etc.
- *device*: If we have done a local attachment this will hold a dictionary with all the attachment information, such as the *path*, the *type*, the *scsi_wwn*, etc.
- *path*: String with the path of the system device that has been created when the volume was attached.

Local attach

Once we have created a volume with *cinderlib* doing a local attachment is really simple, we just have to call the *attach* method from the *Volume* and well get the *Connection* information from the attached volume, and once we are done we call the *detach* method on the *Volume*.

```
vol = lvm.create_volume(size=1)
attach = vol.attach()
with open(attach.path, 'w') as f:
    f.write('*' * 100)
vol.detach()
```

This *attach* method will take care of everything, from gathering our local connection information, to exporting the volume, initializing the connection, and finally doing the local attachment of the volume to our host.

The *detach* operation works in a similar way, but performing the exact opposite steps and in reverse. It will detach the volume from our host, terminate the connection, and if there are no more connections to the volume it will also remove the export of the volume.

> **Attention:** The *Connection* instance returned by the *Volume attach* method also has a *detach* method, but this one behaves differently than the one weve seen in the *Volume*, as it will just perform the local detach step and not the termiante connection or the remove export method.

Remote connection

For a remote connection, where you dont have the driver configuration or access to the management storage network, attaching and detaching volumes is a little more inconvenient, and how you do it will depend on whether you have access to the metadata persistence storage or not.

In any case the general attach flow looks something like this:

- Consumer gets connector information from its host.
- Controller receives the connector information from the consumer. - Controller exports and maps the volume using the connector information and gets the connection information needed to attach the volume on the consumer.
- The consumer gets the connection information. - The consumer attaches the volume using the connection information.

With access to the metadata persistence storage

In this case things are easier, as you can use the persistence storage to pass information between the consumer and the controller node.

Assuming you have the following variables:

- *persistence_config* configuration of your metadata persistence storage.
- *node_id* unique string identifier for your consumer nodes that doesnt change between reboots.
- *cinderlib_driver_configuration* is a dictionary with the Cinder backend configuration needed by cinderlib to connect to the storage.
- *volume_id* ID of the volume we want to attach.

The consumer node must store its connector properties on start using the key-value storage provided by the persistence plugin:

```
import socket
import cinderlib as cl

cl.setup(persistence_config=persistence_config)

kv = cl.Backend.persistence.get_key_values(node_id)
if not kv:
    storage_nw_ip = socket.gethostbyname(socket.gethostname())
    connector_dict = cl.get_connector_properties('sudo', storage_nw_ip,
```

(continues on next page)

(continued from previous page)

```
                                                    True, False)
    value = json.dumps(connector_dict, separators=(',', ':'))
    kv = cl.KeyValue(node_id, value)
    cl.Backend.persistence.set_key_value(kv)
```

Then when we want to attach a volume to *node_id* the controller can retrieve this information using the persistence plugin and export and map the volume for the specific host.

```
import cinderlib as cl

cl.setup(persistence_config=persistence_config)
storage = cl.Backend(**cinderlib_driver_configuration)

kv = cl.Backend.persistence.get_key_values(node_id)
if not kv:
    raise Exception('Unknown node')
connector_info = json.loads(kv[0].value)
vol = storage.Volume.get_by_id(volume_id)
vol.connect(connector_info, attached_host=node_id)
```

Once the volume has been exported and mapped, the connection information is automatically stored by the persistence plugin and the consumer host can attach the volume:

```
vol = storage.Volume.get_by_id(volume_id)
connection = vol.connections[0]
connection.attach()
print('Volume %s attached to %s' % (vol.id, connection.path))
```

When attaching the volume the metadata plugin will store changes to the Connection instance that are needed for the detaching.

No access to the metadata persistence storage

This is more inconvenient, as youll have to handle the data exchange manually as well as the *OS-Brick* library calls to do the attach/detach.

First we need to get the connector information on the host that is going to do the attach:

```
from os_brick.initiator import connector

connector_dict = connector.get_connector_properties('sudo', storage_nw_ip,
                                                    True, False)
```

Now we need to pass this connector information dictionary to the controller node. This part will depend on your specific application/system.

In the controller node, once we have the contents of the *connector_dict* variable we can export and map the volume and get the info needed by the consumer:

```
import cinderlib as cl

cl.setup(persistence_config=persistence_config)
storage = cl.Backend(**cinderlib_driver_configuration)

vol = storage.Volume.get_by_id(volume_id)
conn = vol.connect(connector_info, attached_host=node_id)
connection_info = conn.connection_info
```

We have to pass the contents of *connection_info* information to the consumer node, and that node will use it to attach the volume:

```
import os_brick
from os_brick.initiator import connector

connector_dict = connection_info['connector']
conn_info = connection_info['conn']
protocol = conn_info['driver_volume_type']

conn = connector.InitiatorConnector.factory(
    protocol, 'sudo', user_multipath=True,
    device_scan_attempts=3, conn=connector_dict)
device = conn.connect_volume(conn_info['data'])
print('Volume attached to %s' % device.get('path'))
```

At this point we have the *device* variable that needs to be stored for the disconnection, so we have to either store it on the consumer node, or pass it to the controller node so we can save it with the connector info.

Heres an example on how to save it on the controller node:

```
conn = vol.connections[0]
conn.device = device
conn.save()
```

Warning: At the time of this writing this mechanism doesnt support RBD connections, as this support is added by cinderlib itself.

Multipath

If we want to use multipathing for local attachments we must let the *Backend* know when instantiating the driver by passing the *use_multipath_for_image_xfer=True*:

```
import cinderlib

lvm = cinderlib.Backend(
    volume_driver='cinder.volume.drivers.lvm.LVMVolumeDriver',
    volume_group='cinder-volumes',
    target_protocol='iscsi',
```

(continues on next page)

(continued from previous page)

```
    target_helper='lioadm',
    volume_backend_name='lvm_iscsi',
    use_multipath_for_image_xfer=True,
)
```

Extend

The *Connection* object has an *extend* method that will refresh the hosts view of an attached volume to reflect the latest size of the volume and return the new size in bytes.

There is no need to manually call this method for volumes that are locally attached to the node that calls the *Volumes extend* method, since that call takes care of it.

When extending volumes that are attached to nodes other than the one calling the *Volumes extend* method we will need to either detach and re-attach the volume on the host following the mechanisms explained above, or refresh the current view of the volume.

How we refresh the hosts view of an attached volume will depend on how we are attaching the volumes.

With access to the metadata persistence storage

In this case things are easier, just like it was on the *Remote connection*.

Assuming we have a *volume_id* variable with the volume, and *storage* has the *Backend* instance, all we need to do is:

```
vol = storage.Volume.get_by_id(volume_id)
vol.connections[0].extend()
```

No access to the metadata persistence storage

This is more inconvenient, as youll have to handle the data exchange manually as well as the *OS-Brick* library calls to do the extend.

Well need to get the connector information on the host that is going to do the attach. Asuuming the dictionary is available in *connection_info* the code would look like this:

```
from os_brick.initiator import connector

connector_dict = connection_info['connector']
protocol = connection_info['conn']['driver_volume_type']

conn = connector.InitiatorConnector.factory(
    protocol, 'sudo', user_multipath=True,
    device_scan_attempts=3, conn=connector_dict)
conn.extend()
```

Multi attach

Multi attach support has been added to *Cinder* in the Queens cycle, and its not currently supported by *cinderlib*.

Other methods

All other methods available in the *Snapshot* class will be explained in their relevant sections:

- *load* will be explained together with *json*, *jsons*, *dump*, and *dumps* properties, and the *to_dict* method in the *Serialization* section.
- *refresh* will reload the volume from the metadata storage and reload any lazy loadable property that has already been loaded. Covered in the *Serialization* and *Resource tracking* sections.

3.2.6 Serialization

A *Cinder* driver is stateless on itself, but it still requires the right data to work, and thats why the cinder-volume service takes care of storing the state in the DB. This means that *cinderlib* will have to simulate the DB for the drivers, as some operations actually return additional data that needs to be kept and provided in any future operation.

Originally *cinderlib* stored all the required metadata in RAM, and passed the responsibility of persisting this information to the user of the library.

Library users would create or modify resources using *cinderlib*, and then would have to serialize the resources and manage the storage of this information. This allowed referencing those resources after exiting the application and in case of a crash.

Now we support *Metadata Persistence* plugins, but there are still cases were well want to serialize the data:

- When logging or debugging resources.
- When using a metadata plugin that stores the data in memory.
- Over the wire transmission of the connection information to attach a volume on a remote nodattach a volume on a remote node.

We have multiple methods to satisfy these needs, to serialize the data (*json*, *jsons*, *dump*, *dumps*), to deserialize it (*load*), and to convert to a user friendly object (*to_dict*).

To JSON

We can get a JSON representation of any *cinderlib* object - *Backend*, *Volume*, *Snapshot*, and *Connection* - using their following properties:

- *json*: Returns a JSON representation of the current object information as a Python dictionary. Lazy loadable objects that have not been loaded will not be present in the resulting dictionary.
- *jsons*: Returns a string with the JSON representation. Its the equivalent of converting to a string the dictionary from the *json* property.
- *dump*: Identical to the *json* property with the exception that it ensures all lazy loadable attributes have been loaded. If an attribute had already been loaded its contents will not be refreshed.

- *dumps*: Returns a string with the JSON representation of the fully loaded object. Its the equivalent of converting to a string the dictionary from the *dump* property.

Besides these resource specific properties, we also have their equivalent methods at the library level that will operate on all the *Backends* present in the application.

> **Attention:** On the objects, these are properties (*volume.dumps*), but on the library, these are methods (*cinderlib.dumps()*).

Note: We dont have to worry about circular references, such as a *Volume* with a *Snapshot* that has a reference to its source *Volume*, since *cinderlib* is prepared to handle them.

To demonstrate the serialization in *cinderlib* we can look at an easy way to save all the *Backends* resources information from an application that uses *cinderlib* with the metadata stored in memory:

```
with open('cinderlib.txt', 'w') as f:
    f.write(cinderlib.dumps())
```

In a similar way we can also store a single *Backend* or a single *Volume*:

```
vol = lvm.create_volume(size=1)

with open('lvm.txt', 'w') as f:
    f.write(lvm.dumps)

with open('vol.txt', 'w') as f:
    f.write(vol.dumps)
```

We must remember that *dump* and *dumps* triggers loading of properties that are not already loaded. Any lazy loadable property that was already loaded will not be updated. A good way to ensure we are using the latest data is to trigger a *refresh* on the backends before doing the *dump* or *dumps*.

```
for backend in cinderlib.Backend.backends:
    backend.refresh()

with open('cinderlib.txt', 'w') as f:
    f.write(cinderlib.dumps())
```

When serializing *cinderlib* resources well get all the data currently present. This means that when serializing a volume that is attached and has snapshots well get them all serialized.

There are some cases where we dont want this, such as when implementing a persistence metadata plugin. We should use the *to_json* and *to_jsons* methods for such cases, as they will return a simplified serialization of the resource containing only the data from the resource itself.

From JSON

Just like we had the *json*, *jsons*, *dump*, and *dumps* methods in all the *cinderlib* objects to serialize data, we also have the *load* method to deserialize this data back and recreate a *cinderlib* internal representation from JSON, be it stored in a Python string or a Python dictionary.

The *load* method is present in *Backend*, *Volume*, *Snapshot*, and *Connection* classes as well as in the library itself. The resource specific *load* class method is the exact counterpart of the serialization methods, and it will deserialize the specific resource from the class its being called from.

The librarys *load* method is capable of loading anything we have serialized. Not only can it load the full list of *Backends* with their resources, but it can also load individual resources. This makes it the recommended way to deserialize any data in *cinderlib*. By default, serialization and the metadata storage are disconnected, so loading serialized data will not ensure that the data is present in the persistence storage. We can ensure that deserialized data is present in the persistence storage passing *save=True* to the loading method.

Considering the files we created in the earlier examples we can easily load our whole configuration with:

```
# We must have initialized the Backends before reaching this point

with open('cinderlib.txt', 'r') as f:
    data = f.read()
backends = cinderlib.load(data, save=True)
```

And for a specific backend or an individual volume:

```
# We must have initialized the Backends before reaching this point

with open('lvm.txt', 'r') as f:
    data = f.read()
lvm = cinderlib.load(data, save=True)

with open('vol.txt', 'r') as f:
    data = f.read()
vol = cinderlib.load(data)
```

This is the preferred way to deserialize objects, but we could also use the specific objects *load* method.

```
# We must have initialized the Backends before reaching this point

with open('lvm.txt', 'r') as f:
    data = f.read()
lvm = cinderlib.Backend.load(data)

with open('vol.txt', 'r') as f:
    data = f.read()
vol = cinderlib.Volume.load(data)
```

To dict

Serialization properties and methos presented earlier are meant to store all the data and allow reuse of that data when using drivers of different releases. So it will include all required information to be backward compatible when moving from release N *Cinder* drivers to release N+1 drivers.

There will be times when well just want to have a nice dictionary representation of a resource, be it to log it, to display it while debugging, or to send it from our controller application to the node where we are going to be doing the attachment. For these specific cases all resources, except the *Backend* have a *to_dict* method (not property this time) that will only return the relevant data from the resources.

Backend configuration

When *cinderlib* serializes any object it also stores the *Backend* this object belongs to. For security reasons it only stores the identifier of the backend by default, which is the *volume_backend_name*. Since we are only storing a reference to the *Backend*, this means that when we are going through the deserialization process the *Backend* the object belonged to must already be present in *cinderlib*.

This should be OK for most *cinderlib* usages, since its common practice to store the storage backend connection information (credentials, addresses, etc.) in a different location than the data; but there may be situations (for example while testing) where well want to store everything in the same file, not only the *cinderlib* representation of all the storage resources but also the *Backend* configuration required to access the storage array.

To enable the serialization of the whole driver configuration we have to specify *output_all_backend_info=True* on the *cinderlib* initialization resulting in a self contained file with all the information required to manage the resources.

This means that with this configuration option we wont need to configure the *Backends* prior to loading the serialized JSON data, we can just load the data and *cinderlib* will automatically setup the *Backends*.

3.2.7 Resource tracking

Cinderlib users will surely have their own variables to keep track of the *Backends*, *Volumes*, *Snapshots*, and *Connections*, but there may be cases where this is not enough, be it because we are in a place in our code where we dont have access to the original variables, because we want to iterate all instances, or maybe we are running some manual tests and we have lost the reference to a resource.

For these cases we can use *cinderlibs* various tracking systems to access the resources. These tracking systems are also used by *cinderlib* in the serialization process. They all used to be in memory, but some will now reside in the metadata persistence storage.

Cinderlib keeps track of all:

- Initialized *Backends*.
- Existing volumes in a *Backend*.
- Connections to a volume.
- Local attachment to a volume.
- Snapshots for a given volume.

Initialized *Backends* are stored in a dictionary in *Backends.backends* using the *volume_backend_name* as key.

Existing volumes in a *Backend* are stored in the persistence storage, and can be lazy loaded using the *Backend* instances *volumes* property.

Existing *Snapshots* for a *Volume* are stored in the persistence storage, and can be lazy loaded using the *Volume* instances *snapshots* property.

Connections to a *Volume* are stored in the persistence storage, and can be lazy loaded using the *Volume* instances *connections* property.

Note: Lazy loadable properties will only load the value the first time we access them. Successive accesses will just return the cached value. To retrieve latest values for them as well as for the instance we can use the *refresh* method.

The local attachment *Connection* of a volume is stored in the *Volume* instances *local_attach* attribute and is stored in memory, so unloading the library will lose this information.

We can easily use all these properties to display the status of all the resources weve created:

```
# If volumes lazy loadable property was already loaded, refresh it
lvm_backend.refresh()

for vol in lvm_backend.volumes:
    print('Volume %s is currently %s' % (vol.id, vol.status)

    # Refresh volume's snapshots and connections if previously lazy loaded
    vol.refresh()

    for snap in vol.snapshots:
        print('Snapshot %s for volume %s is currently %s' %
              (snap.id, snap.volume.id, snap.status))

    for conn in vol.connections:
        print('Connection from %s with ip %s to volume %s is %s' %
              (conn.connector_info['host'], conn.connector_info['ip'],
               conn.volume.id, conn.status))
```

3.2.8 Metadata Persistence

Cinder drivers are not stateless, and the interface between the *Cinder* core code and the drivers allows them to return data that can be stored in the database. Some drivers, that have not been updated, are even accessing the database directly.

Because *cinderlib* uses the *Cinder* drivers as they are, it cannot be stateless either.

Originally *cinderlib* stored all the required metadata in RAM, and passed the responsibility of persisting this information to the user of the library.

Library users would create or modify resources using *cinderlib*, and then serialize the resources and manage the storage of this information themselves. This allowed referencing those resources after exiting the application and in case of a crash.

This solution would result in code duplication across projects, as many library users would end up using the same storage types for the serialized data. Thats when the metadata persistence plugin was introduced

in the code.

With the metadata plugin mechanism we can have plugins for different storages and they can be shared between different projects.

Cinderlib includes 2 types of plugins providing 3 different persistence solutions:

- Memory (the default)
- Database
- Database in memory

Using the memory mechanisms users can still use the JSON serialization mechanism to store the metadata.

Currently we have memory and database plugins. Users can store the data wherever they want using the JSON serialization mechanism or with a custom metadata plugin.

Persistence mechanism must be configured before initializing any *Backend* using the *persistence_config* parameter in the *setup* or *global_setup* methods.

Note: When deserializing data using the *load* method on memory based storage we will not be making this data available using the *Backend* unless we pass *save=True* on the *load* call.

Memory plugin

The memory plugin is the fastest one, but its has its drawbacks. It doesnt provide persistence across application restarts and its more likely to have issues than the database plugin.

Even though its more likely to present issues with some untested drivers, it is still the default plugin, because its the plugin that exposes the raw plugin mechanism and will expose any incompatibility issues with external plugins in *Cinder* drivers.

This plugin is identified with the name *memory*, and here we can see a simple example of how to save everything to the database:

```
import cinderlib as cl

cl.setup(persistence_config={'storage': 'memory'})

lvm = cl.Backend(volume_driver='cinder.volume.drivers.lvm.LVMVolumeDriver',
                 volume_group='cinder-volumes',
                 target_protocol='iscsi',
                 target_helper='lioadm',
                 volume_backend_name='lvm_iscsi')
vol = lvm.create_volume(1)

with open('lvm.txt', 'w') as f:
    f.write(lvm.dumps)
```

And how to load it back:

```
import cinderlib as cl

cl.setup(persistence_config={'storage': 'memory'})

lvm = cl.Backend(volume_driver='cinder.volume.drivers.lvm.LVMVolumeDriver',
                 volume_group='cinder-volumes',
                 target_protocol='iscsi',
                 target_helper='lioadm',
                 volume_backend_name='lvm_iscsi')

with open('cinderlib.txt', 'r') as f:
    data = f.read()
backends = cl.load(data, save=True)
print backends[0].volumes
```

Database plugin

This metadata plugin is the most likely to be compatible with any *Cinder* driver, as its built on top of *Cinders* actual database layer.

This plugin includes 2 storage options: memory and real database. They are identified with the storage identifiers *memory_db* and *db* respectively.

The memory option will store the data as an in memory SQLite database. This option helps debugging issues on untested drivers. If a driver works with the memory database plugin, but doesnt with the *memory* one, then the issue is most likely caused by the driver accessing the database. Accessing the database could be happening directly importing the database layer, or indirectly using versioned objects.

The memory database doesnt require any additional configuration, but when using a real database we must pass the connection information using SQLAlchemy database URLs format as the value of the *connection* key.

```
import cinderlib as cl

persistence_config = {'storage': 'db', 'connection': 'sqlite:///cl.sqlite'}
cl.setup(persistence_config=persistence_config)

lvm = cl.Backend(volume_driver='cinder.volume.drivers.lvm.LVMVolumeDriver',
                 volume_group='cinder-volumes',
                 target_protocol='iscsi',
                 target_helper='lioadm',
                 volume_backend_name='lvm_iscsi')
vol = lvm.create_volume(1)
```

Using it later is exactly the same:

```
import cinderlib as cl

persistence_config = {'storage': 'db', 'connection': 'sqlite:///cl.sqlite'}
cl.setup(persistence_config=persistence_config)
```

(continues on next page)

(continued from previous page)

```
lvm = cl.Backend(volume_driver='cinder.volume.drivers.lvm.LVMVolumeDriver',
                 volume_group='cinder-volumes',
                 target_protocol='iscsi',
                 target_helper='lioadm',
                 volume_backend_name='lvm_iscsi')

print lvm.volumes
```

Custom plugins

The plugin mechanism uses Python entrypoints to identify plugins present in the system. So any module exposing the *cinderlib.persistence.storage* entrypoint will be recognized as a *cinderlib* metadata persistence plugin.

As an example, the definition in *setup.py* of the entrypoints for the plugins included in *cinderlib* is:

```
entry_points={
    'cinderlib.persistence.storage': [
        'memory = cinderlib.persistence.memory:MemoryPersistence',
        'db = cinderlib.persistence.dbms:DBPersistence',
        'memory_db = cinderlib.persistence.dbms:MemoryDBPersistence',
    ],
},
```

But there may be cases were we dont want to create entry points available system wide, and we want an application only plugin mechanism. For this purpose *cinderlib* supports passing a plugin instance or class as the value of the *storage* key in the *persistence_config* parameters.

The instance and class must inherit from the *PersistenceDriverBase* in *cinderlib/persistence/base.py* and implement all the following methods:

- *db*
- *get_volumes*
- *get_snapshots*
- *get_connections*
- *get_key_values*
- *set_volume*
- *set_snapshot*
- *set_connection*
- *set_key_value*
- *delete_volume*
- *delete_snapshot*
- *delete_connection*
- *delete_key_value*

And the *__init__* method is usually needed as well, and it will receive as keyword arguments the parameters provided in the *persistence_config*. The *storage* key-value pair is not included as part of the keyword parameters.

The invocation with a class plugin would look something like this:

```
import cinderlib as cl
from cinderlib.persistence import base

class MyPlugin(base.PersistenceDriverBase):
    def __init__(self, location, user, password):
        ...

persistence_config = {'storage': MyPlugin, 'location': '127.0.0.1',
                      'user': 'admin', 'password': 'nomoresecrets'}
cl.setup(persistence_config=persistence_config)

lvm = cl.Backend(volume_driver='cinder.volume.drivers.lvm.LVMVolumeDriver',
                 volume_group='cinder-volumes',
                 target_protocol='iscsi',
                 target_helper='lioadm',
                 volume_backend_name='lvm_iscsi')
```

Migrating storage

Metadata is crucial for the proper operation of *cinderlib*, as the *Cinder* drivers cannot retrieve this information from the storage backend.

There may be cases where we want to stop using a metadata plugin and start using another one, but we have metadata on the old plugin, so we need to migrate this information from one backend to another.

To achieve a metadata migration we can use methods *refresh*, *dump*, *load*, and *set_persistence*.

An example code of how to migrate from SQLite to MySQL could look like this:

```
import cinderlib as cl

# Setup the source persistence plugin
persistence_config = {'storage': 'db',
                      'connection': 'sqlite:///cinderlib.sqlite'}
cl.setup(persistence_config=persistence_config)

# Setup backends we want to migrate
lvm = cl.Backend(volume_driver='cinder.volume.drivers.lvm.LVMVolumeDriver',
                 volume_group='cinder-volumes',
                 target_protocol='iscsi',
                 target_helper='lioadm',
                 volume_backend_name='lvm_iscsi')

# Get all the data into memory
data = cl.dump()
```

(continues on next page)

(continued from previous page)

```
# Setup new persistence plugin
new_config = {
    'storage': 'db',
    'connection': 'mysql+pymysql://user:password@IP/cinder?charset=utf8'
}
cl.Backend.set_persistence(new_config)

# Load and save the data into the new plugin
backends = cl.load(data, save=True)
```

Auto-generated documentation is also available:

3.2.9 cinderlib package

Subpackages

cinderlib.cmd package

Submodules

cinderlib.cmd.cinder_cfg_to_python module

Generate Python code to initialize cinderlib based on Cinder config file

This tool generates Python code to instantiate backends using a cinder.conf file.

It supports multiple backends as defined in enabled_backends.

This program uses the oslo.config module to load configuration options instead of using configparser directly because drivers will need variables to have the right type (string, list, integer), and the types are defined in the code using oslo.config.

    cinder-cfg-to_python cinder.conf cinderlib-conf.py

If no output is provided it will use stdout, and if we also dont provide an input file, it will default to /etc/cinder/cinder.conf.

convert(*source*, *dest*)

main()

cinderlib.cmd.cinder_to_yaml module

convert(*cinder_source*, *yaml_dest=None*)

Module contents

cinderlib.persistence package

Submodules

cinderlib.persistence.base module

class DB(*persistence_driver*)

Bases: `object`

Replacement for DB access methods.

This will serve as replacement for methods used by:

- Drivers
- OVOs get_by_id and save methods
- DB implementation

Data will be retrieved using the persistence driver we setup.

```
GET_METHODS_PER_DB_MODEL = {cinder.objects.Volume.model:  'volume_get',
cinder.objects.VolumeType.model:  'volume_type_get',
cinder.objects.Snapshot.model:  'snapshot_get',
cinder.objects.QualityOfServiceSpecs.model:  'qos_specs_get'}
```

get_by_id(*context*, *model*, *id*, **args*, ***kwargs*)

classmethod image_volume_cache_get_by_volume_id(*context*, *volume_id*)

qos_specs_get(*context*, *qos_specs_id*, *inactive=False*)

snapshot_get(*context*, *snapshot_id*, **args*, ***kwargs*)

volume_admin_metadata_delete(*context*, *volume_id*, *key*)

volume_get(*context*, *volume_id*, **args*, ***kwargs*)

volume_get_all_by_host(*context*, *host*, *filters=None*)

volume_type_get(*context*, *id*, *inactive=False*, *expected_fields=None*)

class PersistenceDriverBase(***kwargs*)

Bases: `object`

Provide Metadata Persistency for our resources.

This class will be used to store new resources as they are created, updated, and removed, as well as provide a mechanism for users to retrieve volumes, snapshots, and connections.

property db

delete_connection(*connection*)

delete_key_value(*key*)

delete_snapshot(*snapshot*)

delete_volume(*volume*)

get_changed_fields(*resource*)

get_connections(*connection_id=None*, *volume_id=None*)

get_fields(*resource*)

get_key_values(*key*)

get_snapshots(*snapshot_id=None*, *snapshot_name=None*, *volume_id=None*)

get_volumes(*volume_id=None*, *volume_name=None*, *backend_name=None*)

reset_change_tracker(*resource*, *fields=None*)

set_connection(*connection*)

set_key_value(*key_value*)

set_snapshot(*snapshot*)

set_volume(*volume*)

vol_type_to_dict(*volume_type*)

Module contents

setup(*config*)

Setup persistence to be used in cinderlib.

By default memory persistance will be used, but there are other mechanisms available and other ways to use custom mechanisms:

- Persistence plugins: Plugin mechanism uses Python entrypoints under namespace cinderlib.persistence.storage, and cinderlib comes with 3 different mechanisms, memory, dbms, and memory_dbms. To use any of these one must pass the string name in the storage parameter and any other configuration as keyword arguments.
- Passing a class that inherits from PersistenceDriverBase as storage parameter and initialization parameters as keyword arguments.
- Passing an instance that inherits from PersistenceDriverBase as storage parameter.

Submodules

cinderlib.cinderlib module

class Backend(*volume_backend_name*, ***driver_cfg*)

Bases: `object`

Representation of a Cinder Driver.

User facing attributes are:

- __init__
- json
- jsons
- load
- stats
- create_volume
- global_setup
- validate_connector

backends = {}

property config

create_volume(*size*, *name=''*, *description=''*, *bootable=False*, ***kwargs*)

property dump

property dumps

global_initialization = False

classmethod global_setup(*file_locks_path=None*, *root_helper='sudo'*, *suppress_requests_ssl_warnings=True*, *disable_logs=True*, *non_uuid_ids=False*, *output_all_backend_info=False*, *project_id=None*, *user_id=None*, *persistence_config=None*, *fail_on_missing_backend=True*, *host=None*, ***cinder_config_params*)

property id

property json

property jsons

static list_supported_drivers(*output_version=1*)

Returns dictionary with driver classes names as keys.

The output of the method changes from version to version, so we can pass the output_version parameter to specify which version we are expecting.

Version 1: Original output intended for human consumption, where all dictionary values are strings.

Version 2: Improved version intended for automated consumption.

- type is now a dictionary with detailed information
- **Values retain their types, so well no longer get None** or False.

classmethod load(*json_src*, *save=False*)

classmethod load_backend(*backend_data*)

property pool_names

refresh()

classmethod set_persistence(*persistence_config*)

stats(*refresh=False*)

validate_connector(*connector_dict*)

Raise exception if missing info for volumes connect call.

property volumes

volumes_filtered(*volume_id=None*, *volume_name=None*)

setup(*file_locks_path=None*, *root_helper='sudo'*, *suppress_requests_ssl_warnings=True*, *disable_logs=True*, *non_uuid_ids=False*, *output_all_backend_info=False*, *project_id=None*, *user_id=None*, *persistence_config=None*, *fail_on_missing_backend=True*, *host=None*, ***cinder_config_params*)

cinderlib.exception module

exception InvalidPersistence(*name*)

Bases: Exception

exception NotLocal(*name*)

Bases: Exception

cinderlib.objects module

class Connection(**args*, ***kwargs*)

Bases: *Object*, *LazyVolumeAttr*

Cinderlib Connection info that maps to VolumeAttachment.

On Pike we dont have the connector field on the VolumeAttachment ORM instance so we use the connection_info to store everything.

Well have a dictionary:

{conn: connection info
connector: connector dictionary device: result of connect_volume}

SIMPLE_JSON_IGNORE = ('volume',)

attach()

property attached

property backend

property conn_info

classmethod connect(*volume*, *connector*, ***kwargs*)

property connected

property connector

property connector_info

detach(*force=False*, *ignore_errors=False*, *exc=None*)

property device

device_attached(*device*)

disconnect(*force=False*)

extend()

classmethod get_by_id(*connection_id*)

property path

property protocol

save()

class KeyValue(*key=None*, *value=None*)

Bases: object

class LazyVolumeAttr(*volume*)

Bases: object

LAZY_PROPERTIES = ('volume',)

refresh()

property volume

class NamedObject(*backend*, ***fields_data*)

Bases: *Object*

property description

property name

property name_in_storage

class Object(*backend*, ***fields_data*)

Bases: object

Base class for our resource representation objects.

DEFAULT_FIELDS_VALUES = {}

LAZY_PROPERTIES = ()

SIMPLE_JSON_IGNORE = ()

backend_class

alias of *Backend*

property dump

property dumps

property json

property jsons

classmethod load(*json_src*, *save=False*)

static new_uuid()

classmethod setup(*persistence_driver*, *backend_class*, *project_id*, *user_id*, *non_uuid_ids*)

to_dict()

to_json(*simplified=True*)

to_jsons(*simplified=True*)

class Snapshot(*volume*, ***kwargs*)

Bases: *NamedObject*, *LazyVolumeAttr*

DEFAULT_FIELDS_VALUES = {'metadata': {}, 'status': 'creating'}

SIMPLE_JSON_IGNORE = ('volume',)

create()

create_volume(***new_vol_params*)

delete()

classmethod get_by_id(*snapshot_id*)

classmethod get_by_name(*snapshot_name*)

save()

class Volume(*backend_or_vol*, *pool_name=None*, ***kwargs*)

Bases: *NamedObject*

DEFAULT_FIELDS_VALUES = {'admin_metadata': {}, 'attach_status': 'detached', 'glance_metadata': {}, 'metadata': {}, 'project_id': cinder.context.RequestContext.project_id, 'size': 1, 'status': 'creating', 'user_id': cinder.context.RequestContext.user_id}

LAZY_PROPERTIES = ('snapshots', 'connections')

SIMPLE_JSON_IGNORE = ('snapshots', 'volume_attachment')

attach()

cleanup()

clone(***new_vol_attrs*)

connect(*connector_dict*, ***ovo_fields*)

property connections

create()

create_snapshot(*name=''*, *description=''*, ***kwargs*)

delete()

detach(*force=False*, *ignore_errors=False*)

disconnect(*connection*, *force=False*)

extend(*size*)

classmethod get_by_id(*volume_id*)

classmethod get_by_name(*volume_name*)

refresh()

save()

property snapshots

setup(*persistence_driver*, *backend_class*, *project_id*, *user_id*, *non_uuid_ids*)

cinderlib.serialization module

Oslo Versioned Objects helper file.

These methods help with the serialization of Cinderlib objects that uses the OVO serialization mechanism, so we remove circular references when doing the JSON serialization of objects (for example in a Volume OVO it has a snapshot field which is a Snapshot OVO that has a volume back reference), piggy back on the OVOs serialization mechanism to add/get additional data we want.

datetime_to_primitive(*obj*, *attr*, *value*, *visited=None*)

Stringify time in ISO 8601 with subsecond format.

This is the same code as the one used by the OVO DateTime to_primitive but adding the subsecond resolution with the .%f part in strftime call.

This is backward compatible with cinderlib using code that didnt generate subsecond resolution, because the from_primitive code of the OVO field uses oslo_utils.timeutils.parse_isotime which in the end uses iso8601.parse_date, and since the subsecond format is also ISO8601 it is properly parsed.

dict_to_primitive(*self*, *obj*, *attr*, *value*, *visited=None*)

dump()

Convert to Json everything we have in this system.

dumps()

Convert to a Json string everything we have in this system.

field_ovo_to_primitive(*obj*, *attr*, *value*, *visited=None*)

field_to_primitive(*self*, *obj*, *attr*, *value*, *visited=None*)

iterable_to_primitive(*self*, *obj*, *attr*, *value*, *visited=None*)

json()

Convert to Json everything we have in this system.

jsons()

Convert to a Json string everything we have in this system.

load(*json_src*, *save=False*)

Load any json serialized cinderlib object.

obj_from_primitive(*cls*, *primitive*, *context=None*, *original_method=cinder.objects.base.CinderObject.obj_from_primitive*)

obj_to_primitive(*self*, *target_version=None*, *version_manifest=None*, *visited=None*)

setup(*backend_class*)

wrap_to_primitive(*cls*)

cinderlib.utils module

add_by_id(*resource*, *elements*)

find_by_id(*resource_id*, *elements*)

Module contents

3.3 Validated drivers

We are in the process of validating the *cinderlib* support of more *Cinder* drivers and adding more automated testing of drivers on *Cinder*s gate.

For now we have 2 backends, LVM and Ceph, that are tested on every *Cinder* and *cinderlib* patch that is submitted and merged.

We have also been able to manually test multiple backends ourselves and received reports of other backends that have been successfully tested.

In this document we present the list of all these drivers, and for each one we include the storage array that was used, the configuration (with masked sensitive data), any necessary external requirements -such as packages or libraries-, whether it is being automatically tested on the OpenStack gates or not, and any additional notes.

Currently the following backends have been verified:

- *LVM* with LIO
- *Ceph*
- Dell EMC *XtremIO*
- Dell EMC *VMAX*

- *Kaminario* K2
- NetApp *SolidFire*
- HPE *3PAR*
- *Synology*
- *QNAP*

3.3.1 LVM

- *Storage*: LVM with LIO
- *Connection type*: iSCSI
- *Requirements*: None
- *Automated testing*: On *cinderlib* and *Cinder* jobs.

Configuration:

```
backends:
    - volume_backend_name: lvm
      volume_driver: cinder.volume.drivers.lvm.LVMVolumeDriver
      volume_group: cinder-volumes
      target_protocol: iscsi
      target_helper: lioadm
```

3.3.2 Ceph

- *Storage*: Ceph/RBD
- *Versions*: Luminous v12.2.5
- *Connection type*: RBD
- *Requirements*:
  - `ceph-common` package
  - `ceph.conf` file
  - Ceph keyring file
- *Automated testing*: On *cinderlib* and *Cinder* jobs.
- *Notes*:
  - If we dont define the `keyring` configuration parameter (must use an absolute path) in our `rbd_ceph_conf` to point to our `rbd_keyring_conf` file, well need the `rbd_keyring_conf` to be in `/etc/ceph/`.
  - **`rbd_keyring_confg` must always be present and must follow the naming** convention of `$cluster.client.$rbd_user.conf`.
  - Current driver cannot delete a snapshot if theres a dependent volume (a volume created from it exists).

Configuration:

```
backends:
    - volume_backend_name: ceph
      volume_driver: cinder.volume.drivers.rbd.RBDDriver
      rbd_user: cinder
      rbd_pool: volumes
      rbd_ceph_conf: tmp/ceph.conf
      rbd_keyring_conf: /etc/ceph/ceph.client.cinder.keyring
```

3.3.3 XtremIO

- *Storage*: Dell EMC XtremIO
- *Versions*: v4.0.15-20_hotfix_3
- *Connection type*: iSCSI, FC
- *Requirements*: None
- *Automated testing*: No

Configuration for iSCSI:

```
backends:
    - volume_backend_name: xtremio
      volume_driver: cinder.volume.drivers.dell_emc.xtremio.XtremIOISCSIDriver
      xtremio_cluster_name: CLUSTER_NAME
      use_multipath_for_image_xfer: true
      san_ip: w.x.y.z
      san_login: user
      san_password: toomanysecrets
```

Configuration for FC:

```
backends:
    - volume_backend_name: xtremio
      volume_driver: cinder.volume.drivers.dell_emc.xtremio.XtremIOFCDriver
      xtremio_cluster_name: CLUSTER_NAME
      use_multipath_for_image_xfer: true
      san_ip: w.x.y.z
      san_login: user
      san_password: toomanysecrets
```

3.3.4 Kaminario

- *Storage*: Kaminario K2
- *Versions*: VisionOS v6.0.72.10
- *Connection type*: iSCSI
- *Requirements*:
  - `krest` Python package from PyPi

- *Automated testing*: No

Configuration:

```
backends:
    - volume_backend_name: kaminario
      volume_driver: cinder.volume.drivers.kaminario.kaminario_iscsi.
↪KaminarioISCSIDriver
      san_ip: w.x.y.z
      san_login: user
      san_password: toomanysecrets
      use_multipath_for_image_xfer: true
```

3.3.5 SolidFire

- *Storage*: NetApp SolidFire
- *Versions*: Unknown
- *Connection type*: iSCSI
- *Requirements*: None
- *Automated testing*: No

Configuration:

```
backends:
    - volume_backend_name: solidfire
      volume_driver: cinder.volume.drivers.solidfire.SolidFireDriver
      san_ip: w.x.y.z
      san_login: admin
      san_password: toomanysecrets
      sf_allow_template_caching = false
      image_volume_cache_enabled = True
      volume_clear = zero
```

3.3.6 VMAX

- *Storage*: Dell EMC VMAX
- *Versions*: Unknown
- *Connection type*: iSCSI
- *Automated testing*: No

```
size_precision: 2
backends:
    - image_volume_cache_enabled: True
      volume_clear: zero
      volume_backend_name: VMAX_ISCSI_DIAMOND
      volume_driver: cinder.volume.drivers.dell_emc.vmax.iscsi.VMAXISCSIDriver
```

(continues on next page)

(continued from previous page)

```
      san_ip: w.x.y.z
      san_rest_port: 8443
      san_login: user
      san_password: toomanysecrets
      vmax_srp: SRP_1
      vmax_array: 000197800128
      vmax_port_groups: [os-iscsi-pg]
```

3.3.7 3PAR

- *Storage*: HPE 3PAR 8200
- *Versions*: 3.3.1.410 (MU2)+P32,P34,P37,P40,P41,P45
- *Connection type*: iSCSI
- *Requirements*:
  - `python-3parclient>=4.1.0` Python package from PyPi
- *Automated testing*: No
- *Notes*:
  - Features work as expected, but due to a bug in the 3PAR driver the stats test (`test_stats_with_creation_on_3par`) fails.

Configuration:

```
backends:
     - volume_backend_name: 3par
       hpe3par_api_url: https://w.x.y.z:8080/api/v1
       hpe3par_username: user
       hpe3par_password: toomanysecrets
       hpe3par_cpg: [CPG_name]
       san_ip: w.x.y.z
       san_login: user
       san_password: toomanysecrets
       volume_driver: cinder.volume.drivers.hpe.hpe_3par_iscsi.
↪HPE3PARISCSIDriver
       hpe3par_iscsi_ips: [w.x.y2.z2,w.x.y2.z3,w.x.y2.z4,w.x.y2.z4]
       hpe3par_debug: false
       hpe3par_iscsi_chap_enabled: false
       hpe3par_snapshot_retention: 0
       hpe3par_snapshot_expiration: 1
       use_multipath_for_image_xfer: true
```

3.3.8 Synology

- *Storage*: Synology DS916+
- *Versions*: DSM 6.2.1-23824 Update 6
- *Connection type*: iSCSI
- *Requirements*: None
- *Automated testing*: No

Configuration:

```
backends:
      - volume_backend_name: synology
        volume_driver: cinder.volume.drivers.synology.synology_iscsi.
↪SynoISCSIDriver
        iscs_protocol: iscsi
        target_ip_address: synology.example.com
        synology_admin_port: 5001
        synology_username: admin
        synology_password: toomanysecrets
        synology_pool_name: volume1
        driver_use_ssl: true
```

3.3.9 QNAP

- *Storage*: QNAP TS-831X
- *Versions*: 4.3.5.0728
- *Connection type*: iSCSI
- *Requirements*: None
- *Automated testing*: No

Configuration:

```
backends:
      - volume_backend_name: qnap
        volume_driver: cinder.volume.drivers.qnap.QnapISCSIDriver
        use_multipath_for_image_xfer: true
        qnap_management_url: https://w.x.y.z:443
        iscsi_ip_address: w.x.y.z
        qnap_storage_protocol: iscsi
        qnap_poolname: Storage Pool 1
        san_login: admin
        san_password: toomanysecrets
```

3.4 Validating a driver

This is a guide for *Cinder* driver maintainers to validate that their drivers are fully supported by *cinderlib* and therefore by projects like Ember-CSI and oVirt that rely on it for storage backend management.

Validation steps include initial manual validation as well as automatic testing at the gate as part of *Cinder*s 3rd party CI jobs.

3.4.1 With DevStack

There are many ways we can install *cinderlib* for the initial validation phase, such as using pip from master repositories or PyPi or using packaged versions of the project, but the official recommendation is to use DevStack.

We believe that, as a *Cinder* driver maintainer, you will be already familiar with DevStack and know how to configure and use it to work with your storage backend, so this will most likely be the easiest way for you to do an initial validation of the driver.

Cinderlib has a DevStack plugin that automatically installs the library as during the stacking process when running the `./stach.sh` script, so we will be adding this plugin to our `local.conf` file.

To use *cinderlib*s master code we will add the line `enable_plugin cinderlib https://git.openstack.org/openstack/cinderlib` after the `[[local|localrc]]` header the in our normal `local.conf` file that already configures our backend. The result will look like this:

```
[[local|localrc]]
enable_plugin cinderlib https://opendev.org/openstack/cinderlib
```

After adding this we can proceed to run the `stack.sh` script.

Once the script has finished executing we will have *cinderlib* installed from Git in our system and we will also have sample Python code of how to use our backend in *cinderlib* using the same backend configuration that exists in our `cinder.conf`. The sample Python code is generated in file `cinderlib.py` in the same directory as our `cinder.conf` file.

The tool generating the `cinderlib.py` file supports `cinder.conf` files with multiple backends, so theres no need to make any additional changes to your `local.conf` if you usually deploy DevStack with multiple backends.

The generation of the sample code runs at the very end of the stacking process (the `extra` stage), so we can use other DevStack storage plugins, such as the Ceph plugin, and the sample code will still be properly generated.

For the LVM default backend the contents of the `cinderlib.py` file are:

```
$ cat /etc/cinder/cinderlib.py
import cinderlib as cl

lvmdriver_1 = cl.Backend(volume_clear="zero", lvm_type="auto",
                         volume_backend_name="lvmdriver-1",
                         target_helper="lioadm",
                         volume_driver="cinder.volume.drivers.lvm.
↪LVMVolumeDriver",
```

(continues on next page)

(continued from previous page)

```
                            image_volume_cache_enabled=True,
                            volume_group="stack-volumes-lvmdriver-1")
```

To confirm that this automatically generated configuration is correct we can do:

```
$ cd /etc/cinder
$ mv cinderlib.py example.py
$ python
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from pprint import pprint as pp
>>> import cinderlib
>>> pp(example.lvmdriver_1.stats())
{'driver_version': '3.0.0',
 'pools': [{'QoS_support': False,
            'backend_state': 'up',
            'filter_function': None,
            'free_capacity_gb': 4.75,
            'goodness_function': None,
            'location_info': 'LVMVolumeDriver:localhost.localdomain:stack-
↪volumes-lvmdriver-1:thin:0',
            'max_over_subscription_ratio': '20.0',
            'multiattach': True,
            'pool_name': 'lvmdriver-1',
            'provisioned_capacity_gb': 0.0,
            'reserved_percentage': 0,
            'thick_provisioning_support': False,
            'thin_provisioning_support': True,
            'total_capacity_gb': 4.75,
            'total_volumes': 1}],
 'shared_targets': False,
 'sparse_copy_volume': True,
 'storage_protocol': 'iSCSI',
 'vendor_name': 'Open Source',
 'volume_backend_name': 'lvmdriver-1'}
>>>
```

Here the name of the variable is *lvmdriver_1*, but in your case the name will be different, as it uses the `volume_backend_name` from the different driver section in the `cinder.conf` file. One way to see the backends that have been initialized by importing the example code is looking into the *example.cl.Backend.backends* dictionary.

Some people deploy DevStack with the default backend and then manually modify the `cinder.conf` file afterwards and restart the *Cinder* services to use their configuration. This is fine as well, as you can easily recreate the Python code to include you backend using the *cinder-cfg-to-cinderlib-code* tool thats installed with *cinderlib*.

Generating the example code manually can be done like this:

```
$ cinder-cfg-to-cinderlib-code /etc/cinder/cinder.conf example.py
```

Now that we know that *cinderlib* can access our backend we will proceed to run *cinderlib*s functional

tests to confirm that all the operations work as expected.

The functional tests use the contents of the existing `/etc/cinder/cinder.conf` file to get the backend configuration. The functional test runner also supports `cinder.conf` files with multiple backends. Test methods have meaningful names ending in the backend name as per the `volume_backend_name` values in the configuration file.

The functional tests are quite fast, as they usually take about 1 minute to run:

```
$ python -m unittest discover -v cinderlib.tests.functional

test_attach_detach_volume_on_lvmdriver-1 (cinderlib.tests.functional.test_
↪basic.BackendFunctBasic) ... ok
test_attach_detach_volume_via_attachment_on_lvmdriver-1 (cinderlib.tests.
↪functional.test_basic.BackendFunctBasic) ... ok
test_attach_volume_on_lvmdriver-1 (cinderlib.tests.functional.test_basic.
↪BackendFunctBasic) ... ok
test_clone_on_lvmdriver-1 (cinderlib.tests.functional.test_basic.
↪BackendFunctBasic) ... ok
test_create_delete_snapshot_on_lvmdriver-1 (cinderlib.tests.functional.test_
↪basic.BackendFunctBasic) ... ok
test_create_delete_volume_on_lvmdriver-1 (cinderlib.tests.functional.test_
↪basic.BackendFunctBasic) ... ok
test_create_snapshot_on_lvmdriver-1 (cinderlib.tests.functional.test_basic.
↪BackendFunctBasic) ... ok
test_create_volume_from_snapshot_on_lvmdriver-1 (cinderlib.tests.functional.
↪test_basic.BackendFunctBasic) ... ok
test_create_volume_on_lvmdriver-1 (cinderlib.tests.functional.test_basic.
↪BackendFunctBasic) ... ok
test_disk_io_on_lvmdriver-1 (cinderlib.tests.functional.test_basic.
↪BackendFunctBasic) ... ok
test_extend_on_lvmdriver-1 (cinderlib.tests.functional.test_basic.
↪BackendFunctBasic) ... ok
test_stats_on_lvmdriver-1 (cinderlib.tests.functional.test_basic.
↪BackendFunctBasic) ... ok
test_stats_with_creation_on_lvmdriver-1 (cinderlib.tests.functional.test_
↪basic.BackendFunctBasic) ... ok

----------------------------------------------------------------------
Ran 13 tests in 54.179s

OK
```

There are a couple of interesting options we can use when the running functional tests using environmental variables:

- `CL_FTEST_LOGGING`: If set it will enable the *Cinder* code to log to stdout during the testing. Undefined by default, which means no output.
- `CL_FTEST_PRECISION`: Integer value describing how much precision we must use when comparing volume sizes. Due to cylinder sizes some storage arrays dont abide 100% to the requested size of the volume. With this option we can define how many decimals will be correct when testing sizes. A value of 2 means that the backend could create a 1.0015869140625GB volume when we

request a 1GB volume and the tests wouldnt fail. Default is zero, which means that it must be perfect or it will fail.

- `CL_FTEST_CFG`: Location of the configuration file. Defaults to `/etc/cinder/cinder.conf`.
- `CL_FTEST_POOL_NAME`: If our backend has multi-pool support and we have configured multiple pools we can use this parameter to define which pool to use for the functional tests. If not defined it will use the first reported pool.

If we encounter problems while running the functional tests, but the *Cinder* service is running just fine, we can go to the #openstack-cinder IRC channel in OFTC, or send an email to the discuss-openstack mailing list starting the subject with *[cinderlib]*.

3.4.2 Cinder 3rd party CI

Once we have been able to successfully run the functional tests its time to make the CI jobs run them on every patch submitted to *Cinder* to ensure the driver keeps being compatible.

There are multiples ways we can accomplish this:

1. Create a 3rd party CI job listening to *cinderlib* patches
2. Create an additional 3rd party CI job in *Cinder*, similar to the one we already have.
3. Reusing our existing 3rd party CI job making it also run the *cinderlib* functional tests.

Options #1 and #2 require more work, as we have to create new jobs, but they make it easier to know that our driver is compatible with *cinderlib*. Option #3 is the opposite, it is easy to setup, but it doesnt make it so obvious that our driver is supported by *cinderlib*.

Configuration

When reusing existing 3rd party CI jobs, the normal setup will generate a valid configuration file on `/etc/cinder/cinder.conf` and *cinderlib* functional tests will use it by default, so we dont have to do anything, but when running a custom CI job we will have to write the configuration ourselves. Though we dont have to do this dynamically. We can write it once and use it in all the *cinderlib* jobs.

To get our backend configuration file for the functional tests we can:

- Use the `cinder.conf` file from one of your DevStack deployments.
- Manually create a minimal `cinder.conf` file.
- Create a custom YAML file.

We can create the minimal `cinder.conf` file using one generated by DevStack. Having a minimal configuration has the advantage of being easy to read.

For an LVM backend could look like this:

```
[DEFAULT]
enabled_backends = lvm

[lvm]
volume_clear = none
target_helper = lioadm
```

(continues on next page)

(continued from previous page)

```
volume_group = cinder-volumes
volume_driver = cinder.volume.drivers.lvm.LVMVolumeDriver
volume_backend_name = lvm
```

Besides the *INI* style configuration files, we can also use YAML configuration files for the functional tests.

The YAML file has 3 key-value pairs that are of interest to us. Only one of them is mandatory, the other 2 are optional.

- `logs`: Boolean value defining whether we want the *Cinder* code to log to stdout during the testing. Defaults to `false`. Takes precedence over environmental variable `CL_TESTING_LOGGING`.
- *size_precision*: Integer value describing how much precision we must use when comparing volume sizes. Due to cylinder sizes some storage arrays dont abide 100% to the requested size of the volume. With this option we can define how many decimals will be correct when testing sizes. A value of 2 means that the backend could create a 1.0015869140625GB volume when we request a 1GB volume and the tests wouldnt fail. Default is zero, which for us means that it must be perfect or it will fail. Takes precedence over environmental variable `CL_FTEST_PRECISION`.
- *backends*: This is a list of dictionaries, each with the configuration parameters that are set in the backend section of the `cinder.conf` file in *Cinder*. This is a mandatory field.

The same configuration we presented for the LVM backend as a minimal `cinder.conf` file would look like this in the YAML format:

```
logs: false
venv_sudo: false
backends:
    - volume_backend_name: lvm
      volume_driver: cinder.volume.drivers.lvm.LVMVolumeDriver
      volume_group: cinder-volumes
      target_helper: lioadm
      volume_clear: none
```

To pass the location of the configuration file to the functional test runner we must use the `CL_FTEST_CFG` environmental variable to point to the location of our file. If we are using a `cinder.conf` file and we save it in `etc/cinder` then we dont need to pass it to the tests runner, since thats the default location.

Use independent job

Creating new jobs is mostly identical to what you already did for the Cinder job with the difference that here we dont need to do a full DevStack installation, as it would take too long. We only need the *cinderlib*, *Cinder*, and *OS-Brick* projects from master and then run *cinderlib*s functional tests.

As an example heres the Ceph job in the *cinderlib* project that takes approximately 8 minutes to run at the gate. In the `pre-run` phase it starts a Ceph demo container to run a Ceph toy cluster as the backend. Then provides a custom configuration YAML file with the backend configuration:

```
- job:
    name: cinderlib-ceph-functional
    parent: openstack-tox-functional-with-sudo
```

(continues on next page)

(continued from previous page)

```
    required-projects:
      - openstack/os-brick
      - openstack/cinder
    pre-run: playbooks/setup-ceph.yaml
    nodeset: ubuntu-bionic
    vars:
      tox_environment:
        CL_FTEST_CFG: "cinderlib/tests/functional/ceph.yaml"
        CL_FTEST_ROOT_HELPER: sudo
        # These come from great-great-grandparent tox job
        NOSE_WITH_HTML_OUTPUT: 1
        NOSE_HTML_OUT_FILE: nose_results.html
        NOSE_WITH_XUNIT: 1
```

For jobs in the *cinderlib* project you can use the `openstack-tox-functional-with-sudo` parent, but for jobs in the *Cinder* project youll have to call this yourself by calling tox or using the same command we used during our manual testing: `python -m unittest discover -v cinderlib.tests.functional`.

Use existing job

The easiest way to run the *cinderlib* functional tests is to reuse an existing *Cinder* CI job, since we dont need to setup anything. We just need to modify our job to run an additional command at the end.

Running the *cinderlib* functional tests after tempest will only add about 1 minute to the jobs current runtime.

You will need to add `openstack/cinderlib` to the `required-projects` configuration of the Zuul job. This will ensure not only that *cinderlib* is installed, but also that is using the right patch when a patch has cross-repository dependencies.

For example, the LVM lio job called `cinder-tempest-dsvm-lvm-lio-barbican` has the following required projects:

```
required-projects:
  - openstack-infra/devstack-gate
  - openstack/barbican
  - openstack/cinderlib
  - openstack/python-barbicanclient
  - openstack/tempest
  - openstack/os-brick
```

To facilitate running the *cinderlib* functional tests in existing CI jobs the *Cinder* project includes 2 playbooks:

- `playbooks/tempest-and-cinderlib-run.yaml`
- `playbooks/cinderlib-run.yaml`

These 2 playbooks support the `cinderlib_ignore_errors` boolean variable to allow CI jobs to run the functional tests and ignore the results so that *cinderlib* failures wont block patches. You can think of it as running the *cinderlib* tests as non voting. We dont recommend setting it, as it would defeat the

purpose of running the jobs at the gate and the *cinderlib* tests are very consistent and reliable and dont raise false failures.

Which one of these 2 playbook to use depends on how we are defining our CI job. For example the LVM job uses the `cinderlib-run.yaml` job in its run.yaml file, and the Ceph job uses the `tempest-and-cinderlib-run.yaml` as its run job command.

If you are running tempest tests using a custom script you can also add the running of the *cinderlib* tests at the end.

3.4.3 Notes

Additional features

The validation process weve discussed tests the basic functionality, but some *Cinder* drivers have additional functionality such as backend QoS, multi-pool support, and support for extra specs parameters that modify advanced volume characteristics -such as compression, deduplication, and thin/thick provisioning- on a per volume basis.

Cinderlib supports these features, but since they are driver specific, there is no automated testing in *cinderlib*s functional tests; but we can test them manually ourselves using the `extra_specs`, `qos_specs` and `pool_name` parameters in the `create_volume` and `clone` methods.

We can see the list of available pools in multi-pool drivers on the `pool_names` property in the Backend instance.

Configuration options

One of the difficulties in the *Cinder* project is determining which options are valid for a specific driver on a specific release. This is usually handled by users checking the *OpenStack* or vendor documentation, which makes it impossible to automate.

There was a recent addition to the *Cinder* driver interface that allowed drivers to report exactly which configuration options were relevant for them via the `get_driver_options` method.

On the initial patch some basic values were added to the drivers, but we urge all driver maintainers to have a careful look at the values currently being returned and make sure they are returning all relevant options, because this will not only be useful for some *Cinder* installers, but also for projects using *cinderlib*, as they will be able to automatically build GUIs to configure backends and to validate provided parameters. Having incorrect or missing values there will result in undesired behavior in those systems.

3.4.4 Reporting results

Once you have completed the process described in this guide you will have a *Cinder* driver that is supported not only in *OpenStack*, but also by *cinderlib* and its related projects, and it is time to make it visible.

For this you just need to submit a patch to the *cinderlib* project modifying the `doc/source/validated.rst` file with the information from your backend.

The information that must be added to the documentation is:

- *Storage*: The make and model of the hardware used.

- *Versions*: Firmware versions used for the manual testing.
- *Connection type*: iSCSI, FC, RBD, etc. Can add multiple types on the same line.
- *Requirements*: Required packages, Python libraries, configuration files, etc. for the driver to work.
- *Automated testing*: Accepted values are:
  - No
  - On *cinderlib* jobs.
  - On *cinder* jobs.
  - On *cinderlib* and *Cinder* jobs.
- *Notes*: Any additional information relevant for *cinderlib* usage.
- *Configuration*: The contents of the YAML file or the driver section in the `cinder.conf`, with masked sensitive data.

3.5 Limitations

Cinderlib works around a number of issues that were preventing the usage of the drivers by other Python applications, some of these are:

- *Oslo config* configuration loading.
- Cinder-volume dynamic configuration loading.
- Privileged helper service.
- DLM configuration.
- Disabling of cinder logging.
- Direct DB access within drivers.
- *Oslo Versioned Objects* DB access methods such as *refresh* and *save*.
- Circular references in *Oslo Versioned Objects* for serialization.
- Using multiple drivers in the same process.

Being in its early development stages, the library is in no way close to the robustness or feature richness that the Cinder project provides. Some of the more noticeable limitations one should be aware of are:

- Most methods dont perform argument validation so its a classic GIGO library.
- The logic has been kept to a minimum and higher functioning logic is expected to be handled by the caller: Quotas, tenant control, migration, etc.
- Limited test coverage.
- Only a subset of Cinder available operations are supported by the library.

Besides *cinderlibs* own limitations the library also inherits some from *Cinders* code and will be bound by the same restrictions and behaviors of the drivers as if they were running under the standard *Cinder* services. The most notorious ones are:

- Dependency on the *eventlet* library.

- Behavior inconsistency on some operations across drivers. For example you can find drivers where cloning is a cheap operation performed by the storage array whereas other will actually create a new volume, attach the source and new volume and perform a full copy of the data.
- External dependencies must be handled manually. So users will have to take care of any library, package, or CLI tool that is required by the driver.
- Relies on command execution via *sudo* for attach/detach operations as well as some CLI tools.

3.6 So You Want to Contribute

For general information on contributing to OpenStack, please check out the contributor guide to get started. It covers all the basics that are common to all OpenStack projects: the accounts you need, the basics of interacting with our Gerrit review system, how we communicate as a community, etc.

The cinderlib library is maintained by the OpenStack Cinder project. To understand our development process and how you can contribute to it, please look at the Cinder projects general contributors page: http://docs.openstack.org/cinder/latest/contributor/contributing.html

Some cinderlib specific information is below.

3.6.1 cinderlib release model

The OpenStack release model for cinderlib is cycle-with-intermediary. This means that there can be multiple full releases of cinderlib from master during a development cycle. The deliverable type of cinderlib is trailing which means that the final release of cinderlib for a development cycle must occur within 3 months after the official OpenStack coordinated release.

At the time of the final release, the stable branch is cut, and cinderlib releases from that branch follow the normal OpenStack stable release policy.

The primary thing to keep in mind here is that there is a period at the beginning of each OpenStack development cycle (for example, Zed) when the master branch in cinder and os-brick is open for Zed development, but cinderlibs master branch is still being used for Yoga development.

3.6.2 cinderlib development model

Because cinderlib depends on cinder and os-brick, its `tox.ini` file is set up to use cinder and os-brick from source (not from released versions) so that changes in cinder and os-brick are immediately available for testing cinderlib.

We follow this practice both for cinderlib master and for the cinderlib stable branches.

3.6.3 cinderlib tox and zuul configuration maintenance

As mentioned above, cinderlibs release schedule is offset from the OpenStack coordinated release schedule by about 3 months. Thus, once cinder and os-brick have had their final release for a cycle, their master branches become the development branch for the *next* cycle, whereas cinderlibs master branch is still the development branch for the *previous* cycle.

This has an impact on both `tox.ini`, which controls your local development testing environment, and `.zuul.yaml`, which controls cinderlibs CI environment. These files require manual maintenance at two points during each OpenStack development cycle:

1. When the cinder (not cinderlib) master branch opens for n+1 cycle development. This happens when the first release candidate for release n is made and the stable branch for release n is created. At this time, cinderlib master is still being used for release n development, so cinderlib master is out of phase with cinder/os-brick master branch, and we must make adjustments to cinderlib masters `tox.ini` and `.zuul.yaml` files.
2. When the cinderlib release n is made, cinderlib master opens for release n+1 development. Thus, cinderlibs master branch is back in phase with cinder/os-brick master branch, and we must make adjustments to cinderlib masters `tox.ini` and `.zuul.yaml` files.

Although cinderlibs `requirements.txt` file is not used by tox (and hence not by Zuul, either), we must maintain it for people who install cinderlib via pypi. Thus it must be checked for correctness before cinderlib is released.

Throughout this section, well be talking about release n and release n+1. The example well use is n is yoga and n+1 is zed.

cinderlib tox.ini maintenance

The items are listed below in the order youll find them in `tox.ini`.

[testenv]setenv

The environment variable `CINDERLIB_RELEASE` must be set to the name of the release that this is the development branch for.

- What is this used by? Its used by `tools/special_install.sh` to figure out what the appropriate upper-constraints file is.
- When should it be changed? The requirements team has been setting up the redirect for https://releases.openstack.org/constraints/upper/{release} at the beginning of each OpenStack development cycle (that is, when master is Zed development, for example, the url https://releases.openstack.org/constraints/upper/zed redirects to the `upper-constraints.txt` in requirements master). Thus, you should only have to change the value of `CINDERLIB_RELEASE` in cinderlib master at the time it opens for release n+1.

[testenv]deps

- While both the cinder and cinderlib master branches are the development branches for the n release cycle (yoga, for example), the base testenv in `tox.ini` in master should look like this:

```
# Use cinder and os-brick from the appropriate development branch instead␣
↪of
# from PyPi.  Defining the egg name we won't overwrite the package␣
↪installed
# by Zuul on jobs supporting cross-project dependencies (include Cinder␣
↪in
# required-projects).  This allows us to also run local tests against the
# latest cinder/brick code instead of released code.
# NOTE: Functional tests may fail if host is missing bindeps from deps␣
↪projects
deps= -r{toxinidir}/test-requirements.txt
      git+https://opendev.org/openstack/os-brick
      git+https://opendev.org/openstack/cinder
```

- When the coordinated release for cycle n has occurred, cinderlibs `tox.ini` in master must be modified so that cinderlib is being tested against cinder and os-brick from the stable branches for the n release (in this example, stable/yoga):

```
deps = -r{toxinidir}/test-requirements.txt
       git+https://opendev.org/openstack/os-brick@stable/yoga
       git+https://opendev.org/openstack/cinder@stable/yoga
```

- After the n release of cinderlib occurs (and the stable/n branch is cut), all of cinder, os-brick, and cinderlib master branches are all n+1 cycle development branches, so:
  - The base testenv in `tox.ini` in master must be modified to use cinder and os-brick from master for testing, reverting the first code block change above.

[testenv:py{3,36,38}]install_command

Note: the actual list of versions may be different from whats listed in the documentation heading above.

This testenv inherits from the base testenv and is the parent for all the unit tests. At the time cinderlib master opens for release n+1 development, check that all supported python versions for the release are listed between the braces (that is, { and }).

- The tox term for this is Generative section names. See the tox docs for more information and the proper syntax.
- The list of supported python runtimes can be found in the OpenStack governance documentation.
- If the supported python runtimes have changed from the previous release, you may also need to update the `python_requires` and the Programming Language classifiers in cinderlibs `setup.cfg` file.

[testenv:docs]install_command

- The `docs` testenv sets a default value for `TOX_CONSTRAINTS_FILE` as part of the `install_command`. This only needs to be changed at the time cinderlib master opens for release n+1. See the discussion above about setting the value for `CINDERLIB_RELEASE`; the same considerations apply here.

  The `[testenv:docs]install_command` is referred to by the other documentation-like testenvs, so you should only have to change the value of `TOX_CONSTRAINTS_FILE` in one place. (But do a scan of `tox.ini` to be sure, and if you find another, please update this page.)

cinderlib .zuul.yaml maintenance

A few things to note about the cinderlib `.zuul.yaml` file.

- The OpenStack QA team defines templates that can be used for testing. A template defines a set of jobs that are run in the check and the gate, and the QA team takes the responsibility to make sure that the template for a release includes all the appropriate tests.

  We dont use the openstack-python3-{release}-jobs template; instead, we directly configure the jobs that are listed in the template. The reason for this is that during cinderlibs trailing development phase (when cinderlib master is the development branch for release n while cinder and os-brick master is the development branch for release n+1, we need to make sure that zuul installs the correct cinder and os-brick branch to test against. We can do this by specifying an override-checkout for cinder, os-brick, and requirements in the job definitions.

  We need to do this even though the zuul jobs will ultimately call cinderlibs tox.ini, where we have already configured the correct branches to use. Thats because Zuul doesnt simply call tox; it does a bunch of setup work to download packages and configure the environment, and if we dont specifically tell Zuul what branches to use, when we run a job on a cinderlib master patch, Zuul figures that all components are supposed to be installed from their master branch including openstack requirements, which specifies the upper-constraints for the release.

- The QA testing templates are defined here: https://opendev.org/openstack/openstack-zuul-jobs/src/branch/master/zuul.d/project-templates.yaml

  The `openstack-zuul-jobs` repo is not branched, so that file will contain the testing templates for all stable branches for which OpenStack CI is still supported.

  After the cinderlib n release, you will open cinderlib for n+1 development. For example, after the yoga release, you will open cinderlib for zed development. For the reasons outlined above, we wont use the zed template directly, but you need to look at it to see what jobs it includes, and make sure that cinderlibs `.zuul.yaml` uses equivalent jobs in each of the check, gate, and post pipelines.

  - Whats meant by equivalent jobs is best explained by an example. The `openstack-python3-zed-jobs` template contains (among other things) an `openstack-tox-py39` job. We dont use that job directly, but instead have an `cinderlib-tox-py39` job defined in the cinderlib `.zuul.yaml` that has `openstack-tox-py39` as a parent. (If the equivalent job you need doesnt exist, you must create it, using the other jobs as examples.)

    We need these cinderlib-specific jobs for running unit tests in the CI because the tests run using the development versions of cinder and os-brick, not released versions, so we need to tell Zuul that it needs to have the code repositories for cinder and os-brick available. (We

also tell it to have the requirements repo available; it will be needed during cinderlibs cycle-trailing development phase.)

With that background, here are the `.zuul.yaml` maintenance tasks.

- When the coordinated release for cycle n has occurred, the jobs in cinderlibs `.zuul.yaml` in master must be updated to use the n stable branch for each of its sibling projects. Letting n be the Yoga relase, what this means is that the jobs will change from looking like this:

```
- job:
    name: cinderlib-tox-py39
    parent: openstack-tox-py39
    required-projects:
      - name: openstack/os-brick
      - name: openstack/cinder
      - name: openstack/requirements
```

  to looking like this:

```
- job:
    name: cinderlib-tox-py39
    parent: openstack-tox-py39
    required-projects:
      - name: openstack/os-brick
        override-checkout: stable/yoga
      - name: openstack/cinder
        override-checkout: stable/yoga
      - name: openstack/requirements
        override-checkout: stable/yoga
```

  Additionally, instead of running the `os-brick-src-tempest-lvm-lio-barbican` job (which is defined in the os-brick repository), we will need to run a special version of that job will be defined in cinderlibs `.zuul.yaml`. This job should already be defined in the file, and will be named `cinderlib-os-brick-src-tempest-lvm-lio-barbican-{release}`. Verify that the job has the correct branch specified for `override-checkout`, and then configure the `check` and `gate` sections to run this job.

- After the n release of cinderlib, when cinderlib master has become the n+1 development branch and is once again in sync with the master branches of cinder and os-brick:
  - remove the `override-checkout` specification from the `cinderlib-tox-*` job definitions
  - take a look at the n+1 release testing template (as discussed above) and make sure that cinderlib is running the correct jobs for the cycle
  - run `os-brick-src-tempest-lvm-lio-barbican` in the check and gate
  - update the definition for the cinderlib-os-brick-src-tempest-lvm-lio-barbican-{release} job so that it will be ready when you need it later in the cycle.

cinderlib requirements.txt maintenance

- When the coordinated release for cycle n has occurred, cinderlibs `requirements.txt` in master must be updated to use only n deliverables (in this example, yoga):

```
# restrict cinder to the yoga release only
cinder>=20.0.0.0,<21.0.0  # Apache-2.0
# brick upper bound is controlled by yoga/upper-constraints
os-brick>=5.2.0  # Apache-2.0
```

- After the n release of cinderlib, when cinderlib master has become the n+1 development branch, `requirements.txt` can again be updated:
  - Remove the upper bound from cinder.
  - The release team likes to push an early release of os-brick from master early in the development cycle. Check to see if that has happened already, and if so, update the minimum version of os-brick to the latest release and make appropriate adjustments to the comments in the file.