Cinder Library Documentation

Release 0.2.2

Gorka Eguileor

Nov 07, 2018

Contents

Cinder Library is a Python library that allows using Cinder storage drivers not only outside of OpenStack but also outside of Cinder, which means there's no need to run MySQL, RabbitMQ, Cinder API, Scheduler, or Volume services to be able to manage your storage.

CHAPTER 1

Cinder Library

1.1 Introduction

Cinder Library is a Python library that allows using storage drivers provided by Cinder outside of OpenStack and without needing to run the Cinder service, so we don't need Keystone, MySQL, or RabbitMQ services to control our storage.

The library is currently in an early development stage and can be considered as a proof of concept and not a finished product at this moment, so please carefully go over the limitations section to avoid surprises.

Due to the limited access to Cinder backends and time constraints the list of drivers that have been manually tested, and using the existing limited functional tests, are:

- LVM with LIO
- Dell EMC XtremIO
- Dell EMC VMAX
- Kaminario K2
- Ceph/RBD
- NetApp SolidFire

1.2 Features

- Use a Cinder driver without running a DBMS, Message broker, or Cinder services.
- Using multiple simultaneous drivers on the same program.
- Stateless: Support full serialization of objects and context to JSON or string so the state can be restored.

- Metadata persistence plugin mechanism.
- 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

1.3 Demo

1.4 Limitations

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 don't perform argument validation so it's a classic GIGO library.
- The logic has been kept to a minimum and higher functioning logic is expected to be handled by the caller.
- There is no CI, or unit tests for that matter, and certainly nothing so fancy as third party vendor CIs, so things could be broken at any point. We only have some automated, yet limited, functional tests.
- Only a subset of Cinder available operations are supported by the library.
- Access to a small number of storage arrays has limited the number of drivers that have been verified to work with cinderlib.

Besides *cinderlib's* own limitations the library also inherits some from *Cinder's* 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 we'll 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.

CHAPTER 2

Installation

2.1 Stable release

The Cinder Library is an interfacing library that doesn't have any storage driver and expects Cinder drivers to be properly installed in the system to run properly.

2.1.1 Drivers

For Red Hat distributions the recommendation is to use RPMs to install the Cinder drivers instead of using *pip*. If we don't 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-queens
# yum-config-manager --enable openstack-queens
# yum update -y
# yum install -y openstack-cinder
```

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

```
# yum install -y https://repos.fedorapeople.org/repos/openstack/openstack-queens/rdo-
↪release-queens-1.noarch.rpm
# yum-config-manager --enable openstack-queens
# sudo yum update -y
# yum install -y openstack-cinder
```

2.1.2 Library

To install Cinder Library we'll use PyPI, so we'll 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 don't have pip installed, this Python installation guide can guide you through the process.

2.1.3 Container

There is a docker image, in case you prefer trying the library without any installation.

The image is called *akrog/cinderlib:stable*, and we can run Python dirrectly with:

```
$ docker run --name=cinderlib --privileged --net=host -v /etc/iscsi:/etc/iscsi -v /
↪dev:/dev -it akrog/cinderlib:stable python
```

2.2 Latest code

2.2.1 Container

A Docker image is automatically built on every commit to the *master* branch. Running a Python shell with the latest *cinderlib* code is as simple as:

```
$ docker run --name=cinderlib --privileged --net=host -v /etc/iscsi:/etc/iscsi -v /
↪dev:/dev -it akrog/cinderlib python
```

2.2.2 Drivers

If we don't 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+https://github.com/openstack/cinder.git
```

2.2.3 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:

```
# python setup.py install
```

CHAPTER 3

Validated drivers

The *Cinder* project has a large number of storage drivers, and all the drivers have their own CI to validate that they are working as expected.

For *cinderlib* this is more complicated, as we don't have the resources of the *Cinder* project. We rely on contributors who have access to the hardware to test if the storage backend works with *cinderlib*.

Note: If you have access to storage hardware supported by *Cinder* not present in here and you would like to test if *cinderlib* works, please follow the *Validating a driver* section and report your results.

Currently the following backends have been verified:

- *LVM* with LIO
- *Ceph*
- Dell EMC *XtremIO*
- Dell EMC *VMAX*
- *Kaminario* K2
- NetApp *SolidFire*

3.1 LVM

- *Cinderlib version*: v0.1.0, v0.2.0
- *Cinder release*: *Pike*, *Queens*, *Rocky*
- *Storage*: LVM with LIO
- *Connection type*: iSCSI
- *Requirements*: None

- *Tested by*: Gorka Eguileor (geguileo/akrog)

Configuration:

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

3.2 Ceph

- *Cinderlib version*: v0.2.0
- *Cinder release*: *Pike*
- *Storage*: Ceph/RBD
- *Versions*: Luminous v12.2.5
- *Connection type*: RBD
- *Requirements*:
  - *ceph-common* package
  - *ceph.conf* file
  - Ceph keyring file
- *Tested by*: Gorka Eguileor (geguileo/akrog)
- *Notes*:
  - If we don't define the *keyring* configuration parameter (must use an absolute path) in our *rbd_ceph_conf* to point to our *rbd_keyring_conf* file, we'll 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 there's a dependent (a volume created from it exists).

Configuration:

```
logs: false
venv_sudo: true
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 XtremIO

- *Cinderlib version*: v0.1.0, v0.2.0
- *Cinder release*: *Pike*, *Queens*, *Rocky*
- *Storage*: Dell EMC XtremIO
- *Versions*: v4.0.15-20_hotfix_3
- *Connection type*: iSCSI, FC
- *Requirements*: None
- *Tested by*: Gorka Eguileor (geguileo/akrog)

Configuration for iSCSI:

```
logs: false
venv_sudo: true
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:

```
logs: false
venv_sudo: false
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
```

3.4 Kaminario

- *Cinderlib version*: v0.1.0, v0.2.0
- *Cinder release*: *Pike*, *Queens*, *Rocky*
- *Storage*: Kaminario K2
- *Versions*: VisionOS v6.0.72.10
- *Connection type*: iSCSI
- *Requirements*:
  - *krest* Python package from PyPi
- *Tested by*: Gorka Eguileor (geguileo/akrog)

Configuration:

```
logs: false
venv_sudo: true
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
```

3.5 SolidFire

- *Cinderlib version*: v0.1.0 with later patch
- *Cinder release*: *Pike*
- *Storage*: NetApp SolidFire
- *Versions*: Unknown
- *Connection type*: iSCSI
- *Requirements*: None
- *Tested by*: John Griffith (jgriffith/j-griffith)

Configuration:

```
logs: false
venv_sudo: true
backends:
    - volume_backend_name: solidfire
      volume_driver: cinder.volume.drivers.solidfire.SolidFireDriver
      san_ip: 192.168.1.4
      san_login: admin
      san_password: admin_password
      sf_allow_template_caching = false
      image_volume_cache_enabled = True
      volume_clear = zero
```

3.6 VMAX

- *Cinderlib version*: v0.1.0
- *Cinder release*: *Pike*, *Queens*, *Rocky*
- *Storage*: Dell EMC VMAX
- *Versions*: Unknown
- *Connection type*: iSCSI
- *Requirements*:
  - On *Pike* we need file */etc/cinder/cinder_dell_emc_config.xml*.

- *Tested by*: Helen Walsh (walshh)

Configuration for *Pike*:

- *Cinderlib* functional test configuration:

```
logs: false
venv_sudo: false
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.VMAXISCSIDrive
```

- Contents of file */etc/cinder/cinder_dell_emc_config.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<EMC>
  <RestServerIp>w.x.y.z</RestServerIp>
  <RestServerPort>8443</RestServerPort>
  <RestUserName>username</RestUserName>
  <RestPassword>toomanysecrets</RestPassword>
  <Array>000197800128</Array>
  <PortGroups>
    <PortGroup>os-iscsi-pg</PortGroup>
  </PortGroups>
  <SRP>SRP_1</SRP>
  <ServiceLevel>Diamond</ServiceLevel>
  <Workload>none</Workload>
  <SSLVerify>/opt/stack/localhost.domain.com.pem</SSLVerify>
</EMC>
```

Configuration for *Queens* and *Rocky*:

```
venv_sudo: false
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
      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]
```

CHAPTER 4

Usage

Providing a 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 Cinder drivers and *cinderlib* are installed we just have to import the library to start using it:

```
import cinderlib
```

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 array's specific connection configuration so it usually doesn't refer to the whole storage. With a backend we'll 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. That's why we have 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.

For extended information on these topics please refer to their specific sections.

4.1 Initialization

The cinderlib itself doesn't 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 and *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,
                 non_uuid_ids=False, output_all_backend_info=False,
                 project_id=None, user_id=None, persistence_config=None,
                 **log_params):
```

The meaning of the library's configuration options are:

4.1.1 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 process on the same host.
- In all the OpenStack deployment.

Cinderlib doesn't 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 current directory to store all file locks required by the drivers.

4.1.2 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 don't require them and we are running the process with a non passwordless *sudo* user.

Defaults to *sudo*.

4.1.3 suppress_requests_ssl_warnings

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

Defaults to *True*.

4.1.4 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 that's not a given validation will reject any non UUID value.

This configuration option allows us to disable the validation on the IDs, at the user's risk.

Defaults to *False*.

4.1.5 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*.

4.1.6 disable_logs

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

Defaults to *True*.

4.1.7 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*.

4.1.8 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*.

4.1.9 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 it's 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.

4.1.10 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.

4.1.11 other keyword arguments

Any other keyword argument passed to the initialization method will be considered a *Cinder* configuration option and passed directly to all the drivers.

This can be useful to set additional logging configuration like debug log level, or many other advanced features.

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

4.2 Backends

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

Note: While some drivers have been manually validated most drivers have not, so there's 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 don't 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 vendor's 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).

4.2.1 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 we'll 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 we'll 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 Vendor's documentation or to the OpenStack's Cinder volume driver configuration documentation.

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, we'll have to check in the Vendor's 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.

Let's see now initialization examples of some storage backends:

4.2.2 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',
)
```

4.2.3 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',
)
```

4.2.4 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',
    san_password='kaminario_password',
    volume_backend_name='kaminario_iscsi',
)
```

For more configurations refer to the *Validated drivers* section.

4.2.5 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 don't have access to the original variable.

For these situations we can use *cinderlib's* 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))
```

4.2.6 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 Developer's 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.

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

4.2.7 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 won't be able to manage pre-existing resources from the backend, and we won't notice when a resource is removed directly on the backend.

4.2.8 Attributes

The *Backend* class has no attributes of interest besides the *backends* mentioned above and the *id*, *config*, and JSON related properties we'll 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 backend's 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 driver's instance. This is the case with the *initialized* property we accessed in the backends listing example.

4.2.9 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.

4.3 Volumes

The *Volume* class provides the abstraction layer required to perform all operations on an existing volume, which means that there will be volume creation operations that will be invoked from other class instances, since the new volume we want to create doesn't exist yet and we cannot use the *Volume* class to manage it.

4.3.1 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 where 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)
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 Cinder's 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* it's a UUID, but we can choose here.
- *project_id*: String identifier, in *Cinder* it's 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*: In *Cinder* used to store the *host@backend#pool* information, here we can just keep some identification of the process that wrote this.
- *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*.

4.3.2 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 don't, so it's the callers responsibility to delete the snapshots.

Deleting a volume with snapshots doesn't have a defined behavior for *Cinder* drivers, since it's 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.

4.3.3 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 doesn't 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.

Example of creating, extending, and deleting a volume:

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

4.3.4 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.

4.4 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 doesn't exist yet and we cannot use the *Snapshot* class to manage it.

4.4.1 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 they 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 have created a snapshot it will be tracked in the *Volume* instance's *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
```

4.4.2 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)
```

4.4.3 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.

4.5 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 it's 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.

4.5.1 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 we'll 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 we've seen in the *Volume*, as it will just perform the local detach step and not the termiante connection or the remove export method.

4.5.2 Remote connection

For a remote connection it's a little more inconvenient at the moment, since you'll have to manually use the *OS-Brick* library on the host that is going to do the attachment.

Note: THIS SECTION IS INCOMPLETE

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

```
import os_brick

# Retrieve the connection information dictionary
```

Then we have to do the connection

```
    # Create the connection
attach = vol.connect(connector_dict)

# Return the volume connection information
```

```
import os_brick

# Do the attachment
```

4.5.3 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',
    target_helper='lioadm',
    volume_backend_name='lvm_iscsi',
    use_multipath_for_image_xfer=True,
)
```

4.5.4 Multi attach

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

4.5.5 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.

4.6 Serialization

A *Cinder* driver is stateless on itself, but it still requires the right data to work, and that's 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 we'll 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*).

4.6.1 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. It's 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. It's 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 don't 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())
```

4.6.2 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 library's *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 object's *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)
```

4.6.3 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 we'll 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.

4.6.4 Backend configuration

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

This should be OK for most *cinderlib* usages, since it's common practice to store you storage backend connection information (credentials, addresses, etc.) in a different location than your data, but there may be situations (for example while testing) where we'll 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 won't 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*.

4.7 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 don't 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 *cinderlib's* 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* instance's *volumes* property.

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

Connections to a *Volume* are stored in the persistence storage, and can be lazy loaded using the *Volume* instance's *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* instance's *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 we've 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))
```

4.8 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. That's 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 medatada.

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.

4.8.1 Memory plugin

The memory plugin is the fastest one, but it's has its drawbacks. It doesn't provide persistence across application restarts and it's more likely to have issues than the database plugin.

Even though it's more likely to present issues with some untested drivers, it is still the default plugin, because it's 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
```

4.8.2 Database plugin

This metadata plugin is the most likely to be compatible with any *Cinder* driver, as its built on top of *Cinder's* 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 doesn't 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 doesn't 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)

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
```

4.8.3 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 don't 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')
```

4.8.4 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()

# 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)
```

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/akrog/cinderlib/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Storage backend and configuration used (replacing sensitive information with asterisks).
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues and the *TODO* file for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write tests

We currently lack decent test coverage, so feel free to look into our existing tests to add missing tests, because any test that increases our coverage is more than welcome.

5.1.5 Write Documentation

Cinder Library could always use more documentation, whether as part of the official Cinder Library docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.6 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/akrog/cinderlib/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *cinderlib* for local development.

1. Fork the *cinderlib* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:YOUR_NAME_HERE/cinderlib.git
```

3. Install tox:

```
$ sudo dnf install python2-tox
```

4. Generate a virtual environment, for example for Python 2.7:

```
$ tox --notest -epy27
```

5. Create a branch for local development:

```
 $ git checkout -b name-of-your-bugfix-or-feature

Now you can make your changes locally.
```

6. When you're done making changes, you can check that your changes pass flake8 and unit tests with:

```
 $ tox -eflake8
 $ tox -epy27

Or if you don't want to create a specific environment for flake8 you can run
things directly without tox:
```

```
$ source .tox/py27/bin/activate
$ flake8 cinderlib tests
$ python setup.py test
```

7. Run functional tests at least with the default LVM configuration:

```
 $ tox -efunctional

To run the LVM functional tests you'll need to have the expected LVM VG
ready.  This can be done using the script we have for this purpose (assuming
we are in the *cinderlib* base directory):
```

```
 $ mkdir temp
 $ cd temp
 $ sudo ../tools/lvm-prepare.sh

The default configuration for the functional tests can be found in the
`tests/functional/lvm.yaml` file.  For additional information on this file
format and running functional tests please refer to the
:doc:`validating_backends` section.

And preferably with all the backends you have at your disposal:
```

```
$ CL_FTESTS_CFG=temp/my-test-config.yaml tox -efunctional
```

8. Commit your changes making sure the commit message is descriptive enough, covering the patch changes as well as why the patch might be necessary. The commit message should also conform to the 50/72 rule.

   $ git add . $ git commit

9. Push your branch to GitHub:

```
$ git push origin name-of-your-bugfix-or-feature
```

10. Submit a pull request through the GitHub website.

5.3 LVM Backend

You may not have a fancy storage array, but that doesn't mean that you cannot use *cinderlib*, because you can always the LVM driver. Here we are going to see how to setup an LVM backend that we can use with *cinderlib*.

Before doing anything you need to make sure you have the required package, for Fedora, CentOS, and RHEL this will be the *targetcli* package, and for Ubuntu the *lio-utils* package.

```
$ sudo yum install targetcli
```

Then we'll need to create your "storage backend", which is actually just a file on your normal filesystem. We'll create a 22GB file with only 1MB currently allocated (this is worse for performance, but better for space), and then we'll mount it as a loopback device and create a PV and VG on the loopback device.

```
$ dd if=/dev/zero of=temp/cinder-volumes bs=1048576 seek=22527 count=1
$ sudo lodevice=`losetup --show -f ./cinder-volumes`
$ sudo pvcreate $lodevice
$ sudo vgcreate cinder-volumes $lodevice
$ sudo vgscan --cache
```

There is a script included in the repository that will do all this for us, so we can just call it from the location where we want to create the file:

```
$ sudo tools/lvm-prepare.sh
```

Now we can use this LVM backend in *cinderlib*:

```
import cinderlib as cl
from pprint import pprint as pp

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(size=1)

attach = vol.attach()
pp('Volume %s attached to %s' % (vol.id, attach.path))
vol.detach()

vol.delete()
```

5.4 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/akrog/cinderlib/pull_requests and make sure that the tests pass for all supported Python versions.

5.5 Tips

To run a subset of tests:

```
$ source .tox/py27/bin/activate
$ python -m unittest tests.test_cinderlib.TestCinderlib.test_lib_setup
```

CHAPTER 6

Validating a driver

OK, so you have seen the project and would like to check if the Cinder driver for your storage backend will work with *cinderlib* or not, but don't want to spend a lot of time to do it.

In that case the best way to do it is using our functional tests with a custom configuration file that has your driver configuration.

6.1 The environment

Before we can test anything we'll need to get our environment ready, which will be comprised of three steps:

- Clone the *cinderlib* project:

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

- Create the testing environment which will include the required Cinder code:

```
$ cd cinderlib
$ tox -efunctional --notest
```

- Install any specific packages our driver requires. Some Cinder drivers have external dependencies that need to be manually installed. These dependencies can be Python package or Linux binaries. If it's the former we will need to install them in the testing virtual environment we created in the previous step.

  For example, for the Kaminario backend we need the *krest* Python package, so here's how we would install the dependency.

```
$ source .tox/py27/bin/active
(py27) $ pip install krest
(py27) $ deactivate
```

  To see the Python dependencies for each backend we can check the driver-requirements.txt file from the Cinder project, or in *cinderlib*'s *setup.py* file listed in the *extras* dictionary.

  If we have binary dependencies we can copy them in *.tox/py27/bin* or just install them globally in our system.

6.2 The configuration

Functional test use a YAML configuration file to get the driver configuration as well as some additional parameters for running the tests, with the default configuration living in the *tests/functiona/lvm.yaml* file.

The configuration file currently supports 3 key-value pairs, with only one being mandatory.

- *logs*: Boolean value defining whether we want the Cinder code to log to stdout during the testing. Defaults to *false*.
- *venv_sudo*: Boolean value that instructs the functional tests on whether we want to run with normal *sudo* or with a custom command that ensure that the virtual environment's binaries are also available. This is not usually necessary, but there are some drivers that use binaries installed by a Python package (like the LVM that requires the *cinder-rtstool* from Cinder). This is also necessary if we've installed a binary in the *.tox/py27/bin* directory.
- *size_precision*: Integer value describing how much precision we must use when comparing volume sizes. Due to cylinder sizes some storage arrays don't 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 wouldn't fail. Default is zero, which for us means that it must be perfect or it will fail.
- *backends*: This is a list of dictionaries each with the configuration parameters that are configured in the *cinder.conf* file in Cinder.

The contents of the default configuration, excluding the comments, are:

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

But like the name implies, *backends* can have multiple drivers configured, and the functional tests will run the tests on them all.

For example a configuration file with LVM, Kaminario, and XtremIO backends would look like this:

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

    - volume_backend_name: xtremio
      volume_driver: cinder.volume.drivers.dell_emc.xtremio.XtremIOISCSIDriver
      use_multipath_for_image_xfer: true
      xtremio_cluster_name: CLUSTER
      san_ip: x.x.x.x
      san_login: user
      san_password: password

    - volume_backend_name: kaminario
      volume_driver: cinder.volume.drivers.kaminario.kaminario_iscsi.
↪KaminarioISCSIDriver
```

```
    use_multipath_for_image_xfer: true
    san_ip: x.x.x.y
    san_login: user
    san_password: password
```

6.3 The validation

Now it's time to run the commands, for this we'll use the *tox* command passing the location of our configuration file via environmental variable *CL_FTESTS_CFG*:

```
$ CL_FTEST_CFG=temp/tests.yaml tox -efunctional

functional develop-inst-nodeps: /home/geguileo/code/cinderlib
functional installed: You are using pip version 8.1.2, ...
functional runtests: PYTHONHASHSEED='2093635202'
functional runtests: commands[0] | unit2 discover -v -s tests/functional
test_attach_detach_volume_on_kaminario (tests_basic.BackendFunctBasic) ... ok
test_attach_detach_volume_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_attach_detach_volume_on_xtremio (tests_basic.BackendFunctBasic) ... ok
test_attach_detach_volume_via_attachment_on_kaminario (tests_basic.BackendFunctBasic)␣
↪... ok
test_attach_detach_volume_via_attachment_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_attach_detach_volume_via_attachment_on_xtremio (tests_basic.BackendFunctBasic) ..
↪. ok
test_attach_volume_on_kaminario (tests_basic.BackendFunctBasic) ... ok
test_attach_volume_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_attach_volume_on_xtremio (tests_basic.BackendFunctBasic) ... ok
test_clone_on_kaminario (tests_basic.BackendFunctBasic) ... ok
test_clone_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_clone_on_xtremio (tests_basic.BackendFunctBasic) ... ok
test_connect_disconnect_multiple_times_on_kaminario (tests_basic.BackendFunctBasic) ..
↪. ok
test_connect_disconnect_multiple_times_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_connect_disconnect_multiple_times_on_xtremio (tests_basic.BackendFunctBasic) ...␣
↪ok
test_connect_disconnect_multiple_volumes_on_kaminario (tests_basic.BackendFunctBasic)␣
↪... ok
test_connect_disconnect_multiple_volumes_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_connect_disconnect_multiple_volumes_on_xtremio (tests_basic.BackendFunctBasic) ..
↪. ok
test_connect_disconnect_volume_on_kaminario (tests_basic.BackendFunctBasic) ... ok
test_connect_disconnect_volume_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_connect_disconnect_volume_on_xtremio (tests_basic.BackendFunctBasic) ... ok
test_create_delete_snapshot_on_kaminario (tests_basic.BackendFunctBasic) ... ok
test_create_delete_snapshot_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_create_delete_snapshot_on_xtremio (tests_basic.BackendFunctBasic) ... ok
test_create_delete_volume_on_kaminario (tests_basic.BackendFunctBasic) ... ok
test_create_delete_volume_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_create_delete_volume_on_xtremio (tests_basic.BackendFunctBasic) ... ok
test_create_snapshot_on_kaminario (tests_basic.BackendFunctBasic) ... ok
test_create_snapshot_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_create_snapshot_on_xtremio (tests_basic.BackendFunctBasic) ... ok
test_create_volume_from_snapshot_on_kaminario (tests_basic.BackendFunctBasic) ... ok
test_create_volume_from_snapshot_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_create_volume_from_snapshot_on_xtremio (tests_basic.BackendFunctBasic) ... ok
```

```
test_create_volume_on_kaminario (tests_basic.BackendFunctBasic) ... ok
test_create_volume_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_create_volume_on_xtremio (tests_basic.BackendFunctBasic) ... ok
test_disk_io_on_kaminario (tests_basic.BackendFunctBasic) ... ok
test_disk_io_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_disk_io_on_xtremio (tests_basic.BackendFunctBasic) ... ok
test_extend_on_kaminario (tests_basic.BackendFunctBasic) ... ok
test_extend_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_extend_on_xtremio (tests_basic.BackendFunctBasic) ... ok
test_stats_on_kaminario (tests_basic.BackendFunctBasic) ... ok
test_stats_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_stats_on_xtremio (tests_basic.BackendFunctBasic) ... ok
test_stats_with_creation_on_kaminario (tests_basic.BackendFunctBasic) ... ok
test_stats_with_creation_on_lvm (tests_basic.BackendFunctBasic) ... ok
test_stats_with_creation_on_xtremio (tests_basic.BackendFunctBasic) ... ok

----------------------------------------------------------------------
Ran 48 tests in x.ys

OK
```

As can be seen each test will have a meaningful name ending in the name of the backend we have provided via the *volume_backend_name* key in the YAML file.

6.4 Reporting results

Once you have run the tests, it's time to report the results so they can be included in the *Validated drivers* section.

To help others use the same backend and help us track how each storage driver was tested please include the following information in your report:

- *Cinderlib* version.
- Storage Array: What hardware and firmware version were used.
- Connection type tested: iSCSI, FC, RBD, etc.
- Dependencies/Requirements for the backend: Packages, Python libraries, configuration files...
- Contents of the YAML file with usernames, passwords, and IPs appropriately masked.
- *Cinder* releases: What cinder releases have been tested.
- Additional notes: Limitations or anything worth mentioning.

To report the results of the tests please create an issue on the project with the information mentioned above and include any errors you encountered if you did encounter any.

CHAPTER 7

Internals

Here we'll go over some of the implementation details within *cinderlib* as well as explanations of how we've resolved the different issues that arise from accessing the driver's directly from outside of the cinder-volume service.

Some of the issues *cinderlib* has had to resolve 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.

CHAPTER 8

Credits

8.1 Development Lead

- Gorka Eguileor <geguileo@redhat.com>

8.2 Contributors

None yet. Why not be the first?

CHAPTER 9

TODO

There are many things that need improvements in *cinderlib*, this is a simple list to keep track of the most relevant topics.

- Connect & attach snapshot for drivers that support it.
- Replication and failover support
- QoS
- Support custom features via extra specs
- Unit tests
- Complete functional tests
- Parameter validation
- Support using *cinderlib* without cinder to just handle the attach/detach
- Add .py examples
- Add support for new Attach/Detach mechanism
- Consistency Groups
- Encryption
- Support name and description attributes in Volume and Snapshot
- Verify multiattach support
- Revert to snapshot support.
- Add documentation to connect remote host. *use_multipath_for_image_xfer* and the *enforce_multipath_for_image_xfer* options.
- Complete internals documentation.
- Document the code.

CHAPTER 10

History

10.1 0.2.3 (2018-MM-DD)

- Bug fixes:
  - Detach a volume when it's unavailable.
- Features:
  - Provide better message when device is not available.

10.2 0.2.2 (2018-07-24)

- Features:
  - Use NOS-Brick to setup OS-Brick for non OpenStack usage.
  - Can setup persistence directly to use key-value storage.
  - Support loading objects without configured backend.
  - Support for Cinder Queens, Rocky, and Master
  - Serialization returns a compact string
- Bug fixes:
  - Workaround for Python 2 getaddrinfo bug
  - Compatibility with requests and requests-kerberos
  - Fix key-value support set_key_value.
  - Fix get_key_value to return KeyValue.
  - Fix loading object without configured backend.

10.3 0.2.1 (2018-06-14)

- Features:
  - Modify fields on connect method.
  - Support setting custom root_helper.
  - Setting default project_id and user_id.
  - Metadata persistence plugin mechanism
  - DB persistence plugin
  - No longer dependent on Cinder's attach/detach code
  - Add device_attached method to update volume on attaching node
  - Support attaching/detaching RBD volumes
  - Support changing persistence plugin after initialization
  - Add saving and refreshing object's metadata
  - Add dump, dumps methods
- Bug fixes:
  - Serialization of non locally attached connections.
  - Accept id field set to None on resource creation.
  - Disabling of sudo command wasn't working.
  - Fix volume cloning on XtremIO
  - Fix iSCSI detach issue related to privsep
  - Fix wrong size in volume from snapshot
  - Fix name & description inconsistency
  - Set created_at field on creation
  - Connection fields not being set
  - DeviceUnavailable exception
  - Multipath settings after persistence retrieval
  - Fix PyPi package created tests module
  - Fix connector without multipath info
  - Always call create_export and remove_export
  - iSCSI unlinking on disconnect

10.4 0.1.0 (2017-11-03)

- First release on PyPI.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search