OS Brick Documentation

Release 6.11.1.dev2

Cinder Contributors

Mar 17, 2025

CONTENTS

os-brick is a Python package containing classes that help with volume discovery and removal from a host.

CHAPTER

ONE

INSTALLATION GUIDE

1.1 Installation

At the command line:

```
$ pip install os-brick
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv os-brick
$ pip install os-brick
```

Or, from source:

```
$ git clone https://opendev.org/openstack/os-brick
$ cd os-brick
$ python setup.py install
```

CHAPTER

TWO

USAGE GUIDE

2.1 Tutorial

This tutorial is intended as an introduction to working with **os-brick**.

2.1.1 Prerequisites

Before we start, make sure that you have the **os-brick** distribution *installed*. In the Python shell, the following should run without raising an exception:

```
>>> import os_brick
```

2.1.2 Configuration

There are some os-brick connectors that use file locks to prevent concurrent access to critical sections of the code.

These file locks use the `oslo.concurrency` `lock_utils` module and require the `lock_path` to be configured with the path where locks should be created.

os-brick can use a specific directory just for its locks or use the same directory as the service using os-brick.

The os-brick specific configuration option is `[os_brick]/lock_path`, and if left undefined it will use the value from `[oslo_concurrency]/lock_path`.

2.1.3 Setup

Once os_brick has been loaded it needs to be initialized, which is done by calling the `os_brick.setup` method with the `oslo.conf` configuration.

It is important that the call to `setup` method happens **after** oslo.config has been properly initialized.

```
from oslo_config import cfg
from cinder import version

CONF = cfg.CONF

def main():
    CONF(sys.argv[1:], project='cinder',
         version=version.version_string())
    os_brick.setup(CONF)
```

2.1.4 Fetch all of the initiator information from the host

An example of how to collect the initiator information that is needed to export a volume to this host.

```
from os_brick.initiator import connector


os_brick.setup(CONF)

# what helper do you want to use to get root access?
root_helper = "sudo"
# The ip address of the host you are running on
my_ip = "192.168.1.1"
# Do you want to support multipath connections?
multipath = True
# Do you want to enforce that multipath daemon is running?
enforce_multipath = False
initiator = connector.get_connector_properties(root_helper, my_ip,
                                               multipath,
                                               enforce_multipath)
```

CHAPTER

THREE

REFERENCE

3.1 API Documentation

The **os-brick** package provides the ability to collect host initiator information as well as discovery volumes and removal of volumes from a host.

3.1.1 os_brick OpenStack Brick library

Sub-modules:

initiator Initiator

Bricks Initiator module.

The initator module contains the capabilities for discovering the initiator information as well as discovering and removing volumes from a host.

Sub-modules:

connector Connector

Brick Connector objects for each supported transport protocol.

The connectors here are responsible for discovering and removing volumes for each of the supported transport protocols.

class os_brick.initiator.connector.**InitiatorConnector**

> **static factory**(*protocol*, *root_helper*, *driver=None*, *use_multipath=False*, *device_scan_attempts=3*, *arch=None*, **args*, ***kwargs*)
>
> Build a Connector object based upon protocol and architecture.

class os_brick.initiator.connector.**ISCSIConnector**(*root_helper: str*, *driver=None*, *execute=None*, *use_multipath: bool = False*, *device_scan_attempts: int = 3*, *transport: str = 'default'*, **args*, ***kwargs*)

> Connector class to attach/detach iSCSI volumes.
>
> **connect_volume**(*connection_properties: dict*) → dict[str, str] | None
>
> Attach the volume to instance_name.

Parameters
connection_properties (*dict*) The valid dictionary that describes all of the target volume attributes.

Returns
dict

connection_properties for iSCSI must include: target_portal(s) - ip and optional port target_iqn(s) - iSCSI Qualified Name target_lun(s) - LUN id of the volume Note that plural keys may be used when use_multipath=True

disconnect_volume(*connection_properties: dict*, *device_info: dict*, *force: bool = False*, *ignore_errors: bool = False*) → None

Detach the volume from instance_name.

Parameters

- **connection_properties** (*dict that must include: target_portal(s) - IP and optional port target_iqn(s) - iSCSI Qualified Name target_lun(s) - LUN id of the volume*) The dictionary that describes all of the target volume attributes.
- **device_info** (*dict*) historical difference, but same as connection_props
- **force** (*bool*) Whether to forcefully disconnect even if flush fails.
- **ignore_errors** (*bool*) When force is True, this will decide whether to ignore errors or raise an exception once finished the operation. Default is False.

class os_brick.initiator.connector.**FibreChannelConnector**(*root_helper: str*, *driver=None*, *execute: str | None = None*, *use_multipath: bool = False*, *device_scan_attempts: int = 3*, **args*, ***kwargs*)

Connector class to attach/detach Fibre Channel volumes.

connect_volume(*connection_properties: dict*) → dict

Attach the volume to instance_name.

Parameters
connection_properties (*dict*) The dictionary that describes all of the target volume attributes.

Returns
dict

connection_properties for Fibre Channel must include: target_wwn - World Wide Name target_lun - LUN id of the volume

disconnect_volume(*connection_properties: dict*, *device_info: dict*, *force: bool = False*, *ignore_errors: bool = False*) → None

Detach the volume from instance_name.

Parameters

- **connection_properties** (*dict*) The dictionary that describes all of the target volume attributes.
- **device_info** (*dict*) historical difference, but same as connection_props

connection_properties for Fibre Channel must include: target_wwn - World Wide Name target_lun - LUN id of the volume

class os_brick.initiator.connector.**LocalConnector**(*root_helper*, *driver=None*, **args*, ***kwargs*)

Connector class to attach/detach File System backed volumes.

connect_volume(*connection_properties: dict*) → dict

Connect to a volume.

Parameters

connection_properties (*dict*) The dictionary that describes all of the target volume attributes. `connection_properties` must include:

- `device_path` - path to the volume to be connected

Returns

dict

disconnect_volume(*connection_properties*, *device_info*, *force=False*, *ignore_errors=False*)

Disconnect a volume from the local host.

Parameters

- **connection_properties** (*dict*) The dictionary that describes all of the target volume attributes.
- **device_info** (*dict*) historical difference, but same as connection_props

class os_brick.initiator.connector.**HuaweiStorHyperConnector**(*root_helper*, *driver=None*, **args*, ***kwargs*)

Connector class to attach/detach SDSHypervisor volumes.

connect_volume(*connection_properties*)

Connect to a volume.

Parameters

connection_properties (*dict*) The dictionary that describes all of the target volume attributes.

Returns

dict

disconnect_volume(*connection_properties*, *device_info*, *force=False*, *ignore_errors=False*)

Disconnect a volume from the local host.

Parameters

- **connection_properties** (*dict*) The dictionary that describes all of the target volume attributes.
- **device_info** (*dict*) historical difference, but same as connection_props

class os_brick.initiator.connectors.nvmeof.**NVMeOFConnector**(*root_helper: str*, *driver: HostDriver | None = None*, *use_multipath: bool = False*, *device_scan_attempts: int = 5*, **args*, ***kwargs*)

Connector class to attach/detach NVMe-oF volumes.

connect_volume(*connection_properties: NVMeOFConnProps*) → dict[str, str]

Attach and discover the volume.

disconnect_volume(*connection_properties: dict*, *device_info: dict[str, str]*, *force: bool = False*, *ignore_errors: bool = False*) → None

Flush the volume.

Disconnect of volumes happens on storage system side. Here we could remove connections to subsystems if no volumes are left. But new volumes can pop up asynchronously in the meantime. So the only thing left is flushing or disassembly of a correspondng RAID device.

Parameters

- **connection_properties** (*dict*) The dictionary that describes all of the target volume attributes as described in connect_volume but also with the device_path key containing the path to the volume that was connected (this is added by Nova).
- **device_info** (*dict*) historical difference, but same as connection_props

extend_volume(*connection_properties: dict[str, str]*) → int

Update an attached volume to reflect the current size after extend

The only way to reflect the new size of an NVMe-oF volume on the host is a rescan, which rescans the whole subsystem. This is a problem on attach_volume and detach_volume, but not here, since we will have at least the namespace we are operating on in the subsystem.

The tricky part is knowing when a rescan has already been completed and the volume size on sysfs is final. The rescan may already have happened before this method is called due to an AER message or we may need to trigger it here.

Scans can be triggered manually with nvme ns-rescan or writing 1 in configfs rescan file, or they can be triggered indirectly when calling the nvme list, nvme id-ns, or even using the nvme admin-passthru command.

Even after getting the new size with any of the NVMe commands above, we still need to wait until this is reflected on the host device, because we cannot return to the caller until the new size is in effect.

If we dont see the new size taking effect on the system after 5 seconds, or if we cannot get the new size with nvme, then we rescan in the latter and in both cases we blindly wait 5 seconds and return whatever size is present.

For replicated volumes, the RAID needs to be extended.

get_volume_paths(*connection_properties: NVMeOFConnProps*, *device_info: dict[str, str] | None = None*) → list[str]

Return paths where the volume is present.

classmethod get_connector_properties(*root_helper*, **args*, ***kwargs*) → dict

The NVMe-oF connector properties (initiator uuid and nqn.)

exception Exceptions

Exceptions for the Brick library.

class os_brick.exception.**BrickException**(*message=None*, ***kwargs*)

Base Brick Exception

To correctly use this class, inherit from it and define a message property. That message will get printfd with the keyword arguments provided to the constructor.

class os_brick.exception.**NotFound**(*message=None*, ***kwargs*)

class os_brick.exception.**Invalid**(*message=None*, ***kwargs*)

class os_brick.exception.**InvalidParameterValue**(*message=None*, ***kwargs*)

class os_brick.exception.**NoFibreChannelHostsFound**(*message=None*, ***kwargs*)

class os_brick.exception.**NoFibreChannelVolumeDeviceFound**(*message=None*, ***kwargs*)

class os_brick.exception.**VolumeDeviceNotFound**(*message=None*, ***kwargs*)

class os_brick.exception.**ProtocolNotSupported**(*message=None*, ***kwargs*)

CHAPTER

FOUR

CONTRIBUTING

4.1 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 os-brick 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