Grenade Docs

Release 0.0.1.dev859

OpenStack Grenade Team

Nov 25, 2025

CONTENTS

CHAPTER

ONE

GRENADE

Grenade is an OpenStack test harness to exercise the upgrade process between releases. It uses DevStack to perform an initial OpenStack install and as a reference for the final configuration. Currently Grenade can upgrade Keystone, Glance, Nova, Neutron, Cinder, Swift, and Ceilometer in their default DevStack configurations.

1.1 Goals

Grenade has the following goals:

- Block unintentional project changes that would break the *Theory of Upgrade*. Most Grenade fails that people hit are of this nature.
- Ensure that upgrading a cloud doesnt do something dumb like delete and recreate all your servers/volumes/networks.
- Be able to grow to support additional upgrade scenarios (like sideways migrations from one configuration to another equivalent configuration)

1.2 Theory of Upgrade

Grenade works under the following theory of upgrade.

- New code should work with old configs

  The upgrade process should not require a config change to run a new release. All config behavior is supposed to be deprecated over a release cycle, so that upon release new code works with the last releases configs. Those configs may create deprecation warnings which need to be addressed before the next release, but they should still work and largely have the same behavior.

- New code should need nothing more than db migrations

  Clearly the release of new code may include new database models. Standard upgrade procedure is to turn off all services that touch the database, run the db migration script, and start with new code.

- Resources created by services before upgrade, should still be there after the system is upgraded

  When upgrading Nova you expect all your VMs to still function during the entire upgrade (whether or not Nova services are up). Taking down the control plane should not take down your VMs.

- Any other required changes on upgrade are an **exception** and must be called out in the release notes.

Grenade supports per release specific upgrade scripts (from-juno, from-kilo). These are designed to support upgrades where additional manual steps are needed for a specific upgrade (i.e. from juno to kilo). These should be used sparingly.

The Grenade core team requires the following before landing these kinds of changes:

- The Release Notes for the release where this will be required clearly specify these manual upgrade steps.
- The PTL for the project in question has signed off on this change.

Note

While we expect the various deployment projects within the OpenStack ecosystem, for example TripleO, Kolla, etc, to read the release notes of each project, it is good practice to communicate any exceptional upgrade changes made to Grenade to those teams directly or at least via the openstack-discuss mailing list.

1.3 Status

Grenade is now running on every patch for projects that support upgrade. Gating Grenade configurations exist for the following in OpenStacks CI system (this is not an exhaustive list):

- A cloud with neutron upgraded between releases
- A cloud with neutron that upgrades all services except nova-compute, thus testing RPC backwards compatibility for rolling upgrades.

1.4 Basic Flow

The grenade.sh script attempts to be reasonably readable, so its worth looking there to see whats really going on. This is the super high level version of what that does.

- get 2 devstacks (base & target)
- install base devstack
- perform some sanity checking (currently tempest smoke) to ensure this is right
- allow projects to create resources that should survive upgrade - see projects/*/resources.sh
- shut down all services
- verify resources are still working during shutdown
- upgrade and restart all services
- verify resources are still working after upgrade
- perform some sanity checking (currently tempest smoke) to ensure everything seems good.

The script skips the first two steps (which take care of setting up the 2 devstack environments and installing the base one) when the value of GRENADE_USE_EXTERNAL_DEVSTACK is set to True.

1.4.1 Terminology

Grenade has two DevStack installs present and distinguished between then as base and target.

- **Base**: The initial install that will be upgraded.
- **Target**: The reference install of target OpenStack (maybe just DevStack)

1.5 Directory Structure

Grenade creates a set of directories for both the base and target OpenStack installation sources and DevStack:

```
$STACK_ROOT
 |- logs                # Grenade logs
 |- save                # Grenade state logs
 |- <base>
 |    |- data           # base data
 |    |- logs           # base DevStack logs
 |    |- devstack
 |    |- images         # cache of downloaded images
 |    |- cinder
 |    |- ...
 |    |- swift
 |- <target>
 |    |- data           # target data
 |    |- logs           # target DevStack logs
 |    |- devstack
 |    |- cinder
 |    |- ...
 |    |- swift
```

1.6 Dependencies

This is a non-exhaustive list of dependencies:

- git
- tox

1.7 Install Grenade

Get Grenade from GitHub in the usual way:

```
git clone https://opendev.org/openstack/grenade
```

1.7.1 Optional: running grenade against a remote target

There is an *optional* setup-grenade script that is useful if you are running Grenade against a remote VM from a local laptop.

Grenade knows how to install the current master branch using the included `setup-grenade` script. The arguments are the hostname of the target system that will run the upgrade testing and the user for the target system:

```
./setup-grenade [testbox [testuser]]
```

If you are running Grenade on the same machine you cloned to, you **do not** need to do this.

1.7.2 Configuration

The Grenade repo and branch used can be changed by adding something like this to `localrc`:

```
GRENADE_REPO=git@github.com:dtroyer/grenade.git
GRENADE_BRANCH=dt-test
```

If you need to configure your local devstacks for your specific environment you can do that by creating `devstack.localrc`. This will get appended to the stub devstack configs for BASE and TARGET.

For instance, specifying interfaces for Nova is a common use of `devstack.localrc`:

```
FLAT_INTERFACE=eth1
VLAN_INTERFACE=eth1
```

1.7.3 Run the Upgrade Testing

```
./grenade.sh
```

Read `grenade.sh` for more details of the steps that happen from here.

CHAPTER

TWO

MODULAR GRENADE ARCHITECTURE

Grenade was originally created to demonstrate some level of upgrade capacity for OpenStack projects. Originally this just included a small number of services.

Proposed new basic flow:

- setup_grenade - all the magic setup involved around err traps and filehandle redirects - setup devstack trees
- setup_base - run stack.sh to build the correct base environment
- verify_base - for project in projects; do verify_project; done
- resources.sh create
- resources.sh verify pre-upgrade
- shutdown - for project in projects; do shutdown; done
- snapshot.sh pre_upgrade (NOT YET IMPLEMENTED)
- resources.sh verify_noapi pre-upgrade
- upgrade
- resources.sh verify post-upgrade
- verify_target
- resources.sh destroy

2.1 Modular Components

Assuming the following tree in target projects:

```
devstack/    - devstack plugin directory
   upgrade/   - upgrade scripts
       settings   - adds settings for the upgrade path
       upgrade.sh
       snapshot.sh - snapshots the state of the service, typically a
          database dump (NOT YET IMPLEMENTED)
       from-juno/ - per release
       within-juno/
       from-kilo/
       within-kilo/
       resources.sh
```

This same modular structure exists in the grenade tree with::

> **grenade/**
>
> > **projects/**
> >
> > > **10_ceilometer/**
> > > settings upgrade.sh

2.2 resources.sh

resources.sh is a per-service resource create / verify / destroy interface. What a service does inside a script is up to them.

You can assume your resource script will only be called if your service is running in an upgrade environment. The script should return zero on success for actions, and nonzero on failure.

2.2.1 Calling Interface

The following is the supported calling interface

- resources.sh early_create

  creates a set of sample resources that should survive very early in the upgrade process. This should only be used for horizontal resources that impact other services, that *have* to be available before they do any of their setup. For instance setup of neutron networks.

  Do not use the phase unless you really know why `create` will not work for you.

- resources.sh create

  creates a set of sample resources that should survive upgrade. Script should exit with a nonzero exit code if any resources could not be created.

  Example: create an instance in nova or a volume in cinder

- resources.sh verify (pre-upgrade|post-upgrade)

  verify that the resources were created. Services are running at this point, and the APIs may be expected to work. The second argument indicates whether we are pre-upgrade or post-upgrade.

  Example: use the nova command to verify that the test instance is still ACTIVE, or the cinder command to verify that the volume is still available.

- resources.sh verify_noapi

  verify that the resources are still present. This is called in the phase where services are stopped, and APIs are expected to not be accessible. Resource verification at this phase my require probing underlying components to make sure nothing has gone awry during service shutdown. The second argument indicates whether we are pre-upgrade or post-upgrade.

  Example: check with libvirt to make sure the instance is actually created and running. Bonus points for being able to ping the instance, or otherwise check its live-ness. With cinder, checking that the LVM volume exists and looks reasonable.

- resources.sh destroy

  Resource scripts should be responsible and cleanup all their resources when asked to destroy.

2.2.2 Calling Sequence

The calling sequence during a grenade run looks as follows:

- # start old side
- create (create will be called during the working old side)
- verify pre-upgrade
- # shutdown all services
- verify_noapi pre-upgrade
- # upgrade and start all services
- verify post-upgrade
- destroy

The important thing to remember is verify/verify_noapi will be called multiple times, with multiple different versions of OpenStack. Those phases of the script must not be rerunnable multiple times.

While create / destroy are only going to be called once in the current interface, bonus points for also making those idempotent for resiliancy in testing.

Per-release upgrade scripts

There are times when *exceptional* manual upgrade steps must be performed to get from one release to the next, or even within the same release. Grenade supports this with per-release scripts found in each project, e.g.:

```
projects/
    60_nova/
        from-ocata/
            upgrade-nova
```

Regarding the sequence of when these per-release scripts are called, any `within-$base` script should be run *before* installing new code, and any `from-$base` script should be run *after* installing new code but before starting the services with the new code. This is because configuration or database changes may be needed before the upgraded code is started.

2.2.3 Supporting Methods

In order to assist with the checks listed the following functions exist:

```
resource_save project key value
resource_get project key
```

This allow resource scripts to have memory, and keep track of things like the allocated IP addresses, IDs, and other non deterministic data that is returned from OpenStack API calls.

2.2.4 Environment

Resource scripts get called in a specific environment already set:

- TOP_DIR - will be set to the root of the devstack directory for the BASE version of devstack incase this is needed to find files like a working `openrc`
- GRENADE_DIR - the root directory of the grenade directory.

The following snippet will give you access to both the grenade and TARGET devstack functions:

```
source $GRENADE_DIR/grenaderc
source $GRENADE_DIR/functions
```

2.2.5 Best Practices

Do as many actions as non admin as possible. As early as you can in your resource script its worth allocating a user/project for the script to run as. This ensures isolation against other scripts, and ensures that actions dont only work because admin gets to bypass safeties.

Test side effects, not just API actions. The point of these resource survival scripts is to test that things created beyond the API / DB interaction still work later. Just testing that data can be stored / retrieved from the database isnt very interesting, and should be covered other places. The value in the resource scripts is these side effects. Actual VMs running, actual iscsi targets running, etc. And ensuring these things are not disrupted when the control plane is shifted out from under them.

2.3 Out of Tree Plugins

A grenade plugin can be hosted out of tree in a project tree, similar to external devstack plugins. There are a few subtle differences when this happens.

The plugin structure will live under `$project/devstack/upgrade/` directory.

The plugin is enabled by adding:

```
enable_grenade_plugin <$project> <giturl> [branch]
```

To `pluginrc` in the `GRENADE_DIR`. An additional rc file was required due to sequencing of when plugin functions become available.

Note: when running a job based on the `grenade-base` job, for each devstack plugin defined using the `devstack_plugins`, the corresponding grenade plugin is enabled automatically.

2.3.1 Changing Devstack Localrc

There is also a mechanism that allows a `settings` file change the devstack localrc files with the `devstack_localrc` function.

::

devstack_localrc <base|target> arbitrary stuff to add

Which will take all the rest of the stuff on that line and add it to the localrc for either the base or target devstack.

Please note that `devstack_localrc` only works when grenade performs the configuration of the devstack settings and runs devstack against the base target. When GRENADE_USE_EXTERNAL_DEVSTACK is set to True, as it happens on the Zuul grenade jobs where devstack is configured and executed before grenade, the function has no effect.

2.3.2 Example settings

The following is a reasonable example `settings` for out of tree plugin:

```
register_project_for_upgrade heat
register_db_to_save heat
devstack_localrc base enable_service h-api h-api-cfn h-api-cw h-eng heat
devstack_localrc target enable_service h-api h-api-cfn h-api-cw h-eng heat
```

This registers the project for upgrade, symbolicly enables the heat database for dump during upgrade, and adds the heat services into the service list for base and target.

Its expected that most `settings` files for out of tree plugins will need equivalent lines.

CHAPTER

THREE

GRENADE PLUGIN REGISTRY

Since weve created the external plugin mechanism, its gotten used by a lot of projects. The following is a list of plugins that currently exist. Any project that wishes to list their plugin here is welcomed to.

3.1 Detected Plugins

The following are plugins that a script has found in the openstack/ namespace, which includes but is not limited to official OpenStack projects.

Plugin Name	URL
aodh	https://opendev.org/openstack/aodh
barbican	https://opendev.org/openstack/barbican
ceilometer	https://opendev.org/openstack/ceilometer
cloudkitty	https://opendev.org/openstack/cloudkitty
designate	https://opendev.org/openstack/designate
heat	https://opendev.org/openstack/heat
ironic	https://opendev.org/openstack/ironic
manila	https://opendev.org/openstack/manila
networking-generic-switch	https://opendev.org/openstack/networking-generic-switch
neutron-vpnaas	https://opendev.org/openstack/neutron-vpnaas
octavia	https://opendev.org/openstack/octavia
vitrage	https://opendev.org/openstack/vitrage
watcher	https://opendev.org/openstack/watcher
zaqar	https://opendev.org/openstack/zaqar

CHAPTER

FOUR

GRENADE CODING GUIDE

4.1 General

Grenade is written in POSIX shell script. It specifies BASH and is compatible with Bash 3.

Grenades official repository is located at https://opendev.org/openstack/grenade.

4.2 Scripts

Grenade scripts should generally begin by calling `env(1)` in the shebang line:

```
#!/usr/bin/env bash
```

The script needs to know the location of the Grenade install directory. `GRENADE_DIR` should always point there, even if the script itself is located in a subdirectory:

```
# Keep track of the current grenade directory.
GRENADE_DIR=$(cd $(dirname "$0") && pwd)
```

Many scripts will utilize shared functions from the `functions` file. This file is copied directly from DevStack trunk periodically. There is also an rc file (`grenaderc`) that is sourced to set the default configuration of the user environment:

```
# Keep track of the current grenade directory.
GRENADE_DIR=$(cd $(dirname "$0") && pwd)

# Import common functions
source $GRENADE_DIR/functions

# Import configuration
source $GRENADE_DIR/grenaderc
```

4.3 Documentation

The GitHub repo includes a gh-pages branch that contains the web documentation for Grenade. This is the primary Grenade documentation along with the Grenade scripts themselves.

All of the scripts are processed with shocco to render them with the comments as text describing the script below. For this reason we tend to be a little verbose in the comments _ABOVE_ the code they pertain to. Shocco also supports Markdown formatting in the comments; use it sparingly. Specifically, `grenade.sh` uses Markdown headers to divide the script into logical sections.

CHAPTER

FIVE

STABLE BRANCH TESTING POLICY

Since the Extended Maintenance policy for stable branches was adopted, OpenStack projects are keeping stable branches around after a stable or maintained period, for a phase of indeterminate length called Extended Maintenance. Prior to this resolution, Grenade supported a running voting job down to the oldest+1 stable branch which was supported upstream. Grenade testing on any branch requires prior branch in a working state, DevStack-wise. Due to this requirement and teams resource constraints, Grenade will only provide support for branches in the Maintained phase from the documented Support Phases. This means Grenade testing on oldest Maintained and all Extended Maintenance branches will be made non-voting if they start failing. All other Maintained branches, that is down to the oldest Maintained+1, will be tested and maintained by the Grenade team.

Extended Maintenance team, if any, is always welcome to maintain the Grenade testing on Extended Maintenance branches and the oldest Maintained branch, and make the jobs voting again.

CHAPTER

SIX

CODE

A look at the bits that make it all go

CHAPTER

SEVEN

FOR CONTRIBUTORS

- If you are a new contributor to Grenade please refer: *So You Want to Contribute*

7.1 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.

Below will cover the more project specific information you need to get started with Grenade.

7.1.1 Communication

- IRC channel `#openstack-qa` at OFTC
- Mailing list (prefix subjects with `[qa]` for faster responses) http://lists.openstack.org/cgi-bin/mailman/listinfo/openstack-discuss

7.1.2 Contacting the Core Team

Please refer to the Grenade Core Team contacts.

7.1.3 New Feature Planning

If you want to propose a new feature please read Feature Proposal Process Grenade features are tracked on Launchpad BP.

7.1.4 Task Tracking

We track our tasks in Launchpad.

7.1.5 Reporting a Bug

You found an issue and want to make sure we are aware of it? You can do so on Launchpad. More info about Launchpad usage can be found on OpenStack docs page

7.1.6 Getting Your Patch Merged

All changes proposed to the Grenade require two `Code-Review +2` votes from Grenade core reviewers before one of the core reviewers can approve the patch by giving `Workflow +1` vote.

7.1.7 Project Team Lead Duties

All common PTL duties are enumerated in the PTL guide.

The Release Process for QA is documented in QA Release Process.

7.2 Scripts

- grenade.sh - The main script
- functions - Grenade specific functions
- run_resources.sh
- inc/bootstrap
- inc/plugin
- inc/upgrade
- clean.sh
- cache_git.sh