Neutron Library Documentation

Release 2.6.2.dev2

Neutron development team

Jan 27, 2022

CONTENTS

Neutron-lib is an OpenStack library project used by Neutron, Advanced Services, and third-party projects that aims to provide common functionality across all such consumers. The library is developed with the following goals in mind:

- Decouple sub-projects from Neutron (i.e. no direct neutron imports in sub-projects).
- Pay down Neutron technical debt via refactoring/re-architecting of sub-optimal patterns in their respective neutron-lib implementation.

This document describes the library for contributors of the project, and assumes that you are already familiar with Neutron from an end-user perspective. If not, hop over to the OpenStack doc site

This documentation is generated by the Sphinx toolkit and lives in the source tree. Additional documentation on Neutron and other components of OpenStack can be found on the OpenStack wiki and the *Neutron section of the wiki*. The Neutron Development wiki is also a good resource for new contributors.

Enjoy!

CHAPTER

ONE

INSTALLATION

At the command line:

```
$ pip install neutron-lib
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv neutron-lib
$ pip install neutron-lib
```

CHAPTER

TWO

USAGE

2.1 Hacking Checks

The `neutron_lib.hacking` package implements a number of public flake8 checks intended to help adopters validate their compliance with the latest hacking standards.

To adopt neutron-libs hacking checks:

1. Update your projects `tox.ini` to include hacking checks from neutron-lib. More specifically, copy hacking checks under Checks for neutron and related projects in `[flake8.local-plugin] extension` in neutron-lib `tox.ini` to `[flake8.local-plugin] extension` in your projects `tox.ini`.

   For example in your `tox.ini`:

   ```
   [flake8:local-plugins]
   extension =
     # Checks from neutron-lib
     N521 = neutron_lib.hacking.checks:use_jsonutils
     N524 = neutron_lib.hacking.checks:check_no_contextlib_nested
     N529 = neutron_lib.hacking.checks:no_mutable_default_args
     N530 = neutron_lib.hacking.checks:check_neutron_namespace_imports
     N532 = neutron_lib.hacking.translation_checks:check_log_warn_
   ↪deprecated
     N534 = neutron_lib.hacking.translation_checks:check_raised_
   ↪localized_exceptions
     N536 = neutron_lib.hacking.checks:assert_equal_none
     N537 = neutron_lib.hacking.translation_checks:no_translate_logs
   ```

   Under certain circumstances, adopters may need to ignore specific neutron-lib hacking checks temporarily. You can ignore such checks just by commenting out them (hopefully with a proper reason).

   If your project has its own hacking checks, you can add more rules to `[flake8.local-plugin] extension` along with hacking checks from neutron-lib.

   Note: The above configuration assumes hacking 2.x. If your project uses hacking 1.x, see *Hacking 1.x support* below.

   ---

2. Update your projects `tox.ini` enable any flake8 extensions neutron-libs `tox.ini` does. These are hacking checks otherwise disabled by default that neutron-lib expects to run.

   For example in neutron-libs `tox.ini`:

```
[flake8]
# H904: Delay string interpolations at logging calls
enable-extensions=H904
```

In the example above, adopters should also add `H904` to the `enable-extensions` in their `tox.ini`.

3. Actively adopt neutron-lib hacking checks by running and monitoring the neutron-lib periodic job (as per stadium guidelines and watching for announcements. Announcements regarding neutron-lib adopter hacking checks will be communicated via openstack-discuss email list and neutron meetings.

2.1.1 Hacking 1.x support

If your project uses hacking 1.x, you need a different way to consume hacking checks from neutron-lib.

> **Warning:** hacking 1.x support is deprecated and will be dropped once all neutron related projects migrate to hacking 2.x.

Update your projects `tox.ini` to use `neutron_lib.hacking.checks.factory` for its `local-check-factory`.

For example in your `tox.ini`:

```
[hacking]
local-check-factory = neutron_lib.hacking.checks.factory
```

If your project needs to register additional project specific hacking checks, you can define your own factory function that calls neutron-libs `factory` function.

For example in your projects python source:

```
def my_factory(register):
    # register neutron-lib checks
    neutron_lib_checks.factory(register)
    # register project specific checks
    register(my_project_check)
```

And use your projects own factory in `tox.ini`:

```
[hacking]
local-check-factory = myproject.mypkg.my_factory
```

CHAPTER

THREE

CONTRIBUTOR GUIDE

In the Contributor Guide, you will find information on the Neutron Library components and in how to use them, from a development standpoint.

3.1 Consuming neutron-lib

Many OpenStack projects consume neutron-lib by importing and using its code. As a result, these consumers must define `neutron-lib` as a dependency in their respective `requirements.txt` file. While this is likely nothing new to most OpenStack developers, it may not be obvious to consumers if they need to always use the latest release of neutron-lib, or if they can lag behind using an old(er) release of it.

The answer is that its up to the consuming project if they want to stay current or not in terms of the neutron-lib version they use. While we might like all consumers to stay current and there are benefits to it, this isnt always realistic for projects that have less developers/velocity. Therefore, each project has two options for consuming neutron-lib.

- Opt-in to stay current by adding a comment with `neutron-lib-current` in their `requirements.txt`. This string declares the projects intent to use the latest version of neutron-lib as well as their agreement to stay current with overall OpenStack initiatives. These projects receive updates for free as part of the ongoing neutron-lib work. For more details see the ML archive
- Do not opt-in and rather manage your own consumption of neutron-lib. In this case your project developers must define the version of neutron-lib to use and update the projects code to consume it as they bump up the version. In this scenario most projects will also be managing a back-leveled version of `neutron` since neutron is always current with neutron-lib and might otherwise break the consuming code.

3.2 Neutron-Lib Conventions

3.2.1 Summary

- Standard modules - current cycle + 2 deprecation policy
- Legacy modules - current cycle + 1 deprecation policy
- Private modules - no deprecation warnings of any kind

3.2.2 Interface Contract

The neutron-lib repo exists to provide code with a long-term stable interface for subprojects. If we do need to deprecate something, a debtcollector warning will be added, the neutron-lib core team will make every effort to update any neutron stadium project for you, and you will get at least the current release plus two cycles before it disappears.

In keeping with how hard it is to remove things, the change velocity of this library will be slower than neutron. There are two notable cases where code should go into standard neutron instead of this lib (or your repo):

- It is something common, but changes a lot. An example is something like the neutron Port object. Everyone uses it, but it changes frequently. You dont want to wait for a library release to tweak some neutron feature, and we are not going to force releases quickly because you tried to put it here. Those items will need to be addressed in some other manner (in the case of the Port object, itll be via an auto-magic container object that mimics it.)
- It is something common, but you need it now. Put it in the repo that needs it, get your stuff working. Then consider making it available in the lib, and eventually remove it from your repo when the lib is publishing it. An example would be a new neutron constant. The process would be, put it in neutron along with your change, submit it to the lib, when that constant is eventually available, remove it from neutron and use the lib version.

3.2.3 Private Interfaces

Private interfaces are *PRIVATE*. They will disappear, be renamed, and absolutely no regard will be given to anyone that is using them. They are for internal library use only.

DO NOT USE THEM. THEY WILL CHANGE.

Private interfaces in this library will always have a leading underscore, on the module or function name.

3.3 Review Guidelines

When reviewing neutron-lib changes, please be aware:

- When code is moved from neutron, please evaluate with the following criteria:
  - Is all of the code shared? Dont move neutron-only code.
  - Is the interface good, or does it need to be refactored? If refactoring is required it must be done before the public interface is released to PyPI as once released it must follow our conventions.
  - Does it need new tests, specifically around the interface? We want a global unit coverage greater than 90%, and a per-module coverage greater than 80%. If neutron does not yet have a test, it needs to be added. Note that tests on things like constants are uninteresting, but any code or interface should have a unit test, if you cannot tell for sure that it is not going to be traversed in some alternative way (e.g. tempest/functional coverage).
  - Do the public APIs have their parameters and return values documented using reStructured-Text docstring format (see below)?
  - In certain cases, it may be beneficial to determine how the neutron-lib code changes impact neutron *master*. This can be done as follows:

    * Publish a Do Not Merge dummy patch to neutron that uses the code changes proposed (or already in) neutron-lib. Make sure to mark this neutron change as a DNM (or WIP) and use -1 for workflow to indicate.
    * Publish a change to neutron-lib that uses *Depends-On:* for the dummy change in neutron; this pulls the neutron dummy change into the neutron-lib gate job. For example 386846 uses a dummy neutron-lib patch to test code that already exists in neutron-lib *master* whereas 346554 tests the neutron-lib patchs code itself.
    * View neutron-lib gate job results and repeat as necessary.

- Public APIs should be documented using reST style docstrings that include an overview as well as parameter and return documentation. The format of docstrings can be found in the OpenStack developer hacking docs. Note that public API documentation is a bonus, not a requirement.
- Once public classes and methods are pushed to PyPI as part of a neutron-lib release, they must not be destructively changed without following the full OpenStack deprecation path.

  For example, do not:

  – Change names of classes or methods
  – Reorder method arguments
  – Change side effects

  Alternatives:

  – Add a second method with the new signature
  – Add keyword arguments

  The above implies that if you add something, we are stuck with that interface for a long time, so be careful.
- Removing the code from neutron can be done without a temporary debtcollector notice by following the steps described in the Consume phase of the contributing doc.
- Any code that imports/uses the following python modules should not be moved into neutron-lib:

  – eventlet
- With respect to Oslo config options:

  – Config options should only be included in neutron-lib when the respective functionality that uses the options lives in neutron-lib. In this case the options will need to be exposed as entry points for config generation.
  – Common functionality in neutron-lib that accesses config values should assume the caller has registered them and document such in the docstring for the respective functionality in neutron-lib.

3.4 Contributing

If you would like to contribute to the development of OpenStack, you must follow the steps in this page: https://docs.openstack.org/infra/manual/developers.html

If you already have a good understanding of how the system works and your OpenStack accounts are set up, you can skip to the development workflow section of this documentation to learn how changes to OpenStack should be submitted for review via the Gerrit tool: https://docs.openstack.org/infra/manual/developers.html#development-workflow

Pull requests submitted through GitHub will be ignored.

Bugs should be filed on Launchpad in the neutron project using the `lib` tag, not GitHub: https://bugs.launchpad.net/neutron/+bugs?field.tag=lib

As your code is subject to the review guidelines, please take the time to familiarize yourself with those guidelines.

3.4.1 Rehoming Existing Code

The checklist below aims to provide guidance for developers rehoming (moving) code into neutron-lib. Rehoming approaches that fall outside the scope herein will need to be considered on a case by case basis.

Please note that the effort to rehome existing code from neutron to neutron-lib so that no stadium projects would import directly from neutron has been suspended.

The rehoming workflow procedure has four main phases:

1. *Phase 1: Rehome* the code from neutron into neutron-lib.
2. *Phase 2: Enhance* the code in neutron-lib if necessary.
3. *Phase 3: Release* neutron-lib with the code so consumers can use it.
4. *Phase 4: Consume* by removing the rehomed code from its source and changing references to use neutron-lib.

Phase 1: Rehome

1. Identify the chunk of code for rehoming. Applicable code includes common classes/functions/modules/etc. that are consumed by networking project(s) outside of neutron. Optimal consumption patterns of the code at hand must also be considered to ensure the rehomed code addresses any technical debt. Finally, leave low-hanging fruit for last and tackle the most commonly used code first. If you have any doubt about the applicability of code for rehoming, reach out to one of the neutron core developers before digging in.
2. Find and identify any unit tests for the code being rehomed. These unit tests can often be moved into neutron-lib with minimal effort. After inspecting the applicable unit tests, rewrite any that are non-optimal.
3. Search and understand the consumers of the code being rehomed. This must include other networking projects in addition to neutron itself. At this point it may be determined that the code should be refactored before it is consumed. There are a few common strategies for refactoring, and the one chosen will depend on the nature of the code at hand:

- Refactor/enhance the code as part of the initial neutron-lib patch. If this change will be disruptive to consumers, clearly communicate the change via email list or meeting topic.
- Leave the refactoring to the next (Enhance) phase. In this rehome phase, copy the code as-is into a private module according to our conventions. This approach is slower, but may be necessary in some cases.

4. Understand existing work underway which may impact the rehomed code, for example, in-flight patch sets that update the code being rehomed. In some cases it may make sense to let the in-flight patch merge and solidify a bit before rehoming.
5. Prepare the code for neutron-lib. This may require replacing existing imports with those provided by neutron-lib and/or rewriting/rearchitecting non-optimal code (see above). The interfaces in the rehomed code are subject to our conventions.
6. Prepare the unit test code for neutron-lib. As indicated in the review guidelines we are looking for a high code coverage by tests. This may require adding additional tests if neutron was lacking in coverage.
7. Submit and shepherd your patch through its neutron-lib review. Include a release note that describes the codes old neutron location and new neutron-lib location. Also note that in some cases it makes sense to prototype a change in a consumer project to better understand the impacts of the change, which can be done using the `Depends-On:` approach described in the review guidelines

Examples:

- 319769 brought over a number of common utility functions as-is from neutron into a new package structure within neutron-lib.
- 253661 rehomed neutron callbacks into a private package thats enhanced via 346554.
- 319386 rehomes a validator from neutron into neutron-lib.

Phase 2: Enhance

If the rehomed code is not applicable for enhancements and wasnt made private in Phase 1, you can skip this step.

Develop and shepherd the enhancements to the private rehomed code applicable at this time. Private APIs made public as part of this phase will also need release notes indicating the new public functionality.

Examples:

- 346554 enhances the rehomed private callback API in neutron-lib.

Phase 3: Release

A new neutron-lib release can be cut at any time. You can also request a release by following the README instructions in the openstack/releases project.

Once a release is cut, an openstack infra proposal bot will submit patches to the master branch of all projects that consume neutron-lib to set the new release as the minimum requirement. Someone from the neutron release team can bump *global requirements* (g-r); for example review 393600.

When the bot-proposed requirement patches have merged, your rehomed code can be consumed.

Phase 4: Consume

When code is rehomed from neutron-lib then the original location of the code should be flagged with a debtcollector removal. This will indicate to any consuming projects that the given code is deprecated. Be sure that this change is accompanied by a release note that notes the deprecation.

3.5 Releasing

Before you intend to release a new version of neutron-lib consider posting a sentinel patch that will allow to validate that the neutron-lib hash chosen for tagging is not breaking gate or check jobs affecting a project you care about, first and foremost Neutron.

As the patch shows, upper-constraints must be bypassed and that may itself lead to failures due to upstream package changes. The patch also shows (expected) Grenade failures in that Grenade pulls its own upper-constraints file during the upgrade phase. In general a newer version of neutron-lib is validated through the Tempest -full job (and Grenade runs a subset of it), so Grenade failures can be safely ignored.

In any other case consider (for failures caused by unpinned global requirements) hard-coding a dummy upper-constraints file that itself uses the specific neutron-lib hash you want to test. Furthermore, consider using a commit header that starts with DNM (Do Not Merge) to indicate that the change is just a test, or -2, if you have the right access permissions.

It is also worth noting that every Stadium project will have a periodic job running unit tests and pep8 against the master version of neutron-lib Checking Grafanas periodic dashboard can give you a glimpse into the sanity of the integration between neutron-lib and the Stadium projects, and can be considered the quick check before going ahead with a full blown sentinel patch. Periodic failures can be debugged by viewing the periodic logs

In addition, both the API reference as well as the project docs should be validated to ensure there are no dead links. To do so run `tox -e linkcheck` and address the errors.

3.6 Neutron Lib Internals

3.6.1 API Attributes

Neutrons resource attributes are defined in dictionaries in `api/definitions`.

The map containing all installed resources (for core and active extensions) is in `api/attributes.py`.

Attribute map structure

Example attribute definitions for `dns_name`:

```
'dns_name': {
    'allow_post': True,
    'allow_put': True,
    'default': '',
    'convert_to': convert_to_lowercase,
    'validate': {'type:dns_name': FQDN_MAX_LEN},
    'is_visible': True
},
```

The `validate` item specifies rules for validating the attribute.

The `convert_to` item specifies rules for converting the attribute.

Example attribute definitions for `gateway_ip`:

```
'gateway_ip': {
    'allow_post': True,
    'allow_put': True,
    'default': constants.ATTR_NOT_SPECIFIED,
    'validate': {'type:ip_address_or_none': None},
    'is_visible': True
}
```

Note: a default of `ATTR_NOT_SPECIFIED` indicates that an attribute is not required, but will be generated by the plugin if it is not specified. Particularly, a value of `ATTR_NOT_SPECIFIED` is different from an attribute that has been specified with a value of `None`. For example, if `gateway_ip` is omitted in a request to create a subnet, the plugin will receive `ATTR_NOT_SPECIFIED` and the default gateway IP will be generated. However, if `gateway_ip` is specified as `None`, this means that the subnet does not have a gateway IP.

The following are the defined keys for attribute maps:

`default`	default value of the attribute (if missing, the attribute becomes mandatory)
`allow_post`	the attribute can be used on `POST` requests
`allow_put`	the attribute can be used on `PUT` requests
`validate`	specifies rules for validating data in the attribute
`convert_to`	transformation to apply to the value before it is returned
`convert_list_to`	if the value is a list, apply this transformation to the value before it is returned
`is_filter`	the attribute can be used in `GET` requests as filter
`is_sort_key`	the attribute can be used in `GET` requests as sort_key
`is_visible`	the attribute is returned in `GET` responses
`required_by_policy`	the attribute is required by the policy engine and should therefore be filled by the API layer even if not present in request body
`enforce_policy`	the attribute is actively part of the policy enforcing mechanism, ie: there might be rules which refer to this attribute
`primary_key`	Mark the attribute as a unique key.
`default_overrides_none`	if set, if the value passed is None, it will be replaced by the `default` value
`dict_populate_defaults`	if set, the `default` values of keys inside dict attributes, will be filled if not specified

When extending existing sub-resources, the sub-attribute map must define all extension attributes under the `parameters` object. This instructs the API internals to add the attributes to the existing sub-resource rather than overwrite its existing definition. For example:

```
SUB_RESOURCE_ATTRIBUTE_MAP = {
    'existing_subresource_to_extend': {
        'parameters': {
            'new_attr1': {
                'allow_post': False,
                # etc..
            }
        }
    }
}
```

3.6.2 API Extensions

API extensions provide a standardized way of introducing new API functionality. While the `neutron-lib` project itself does not serve an API, the `neutron` project does and leverages the API extension framework from `neutron-lib`.

API extensions consist of the following high-level constructs:

- API definitions that specify the extensions static metadata. This metadata includes basic details about the extension such as its name, description, alias, etc. as well as its extended resources/sub-resources and required/optional extensions. These definitions live in the `neutron_lib.api.definitions` package.
- API reference documenting the APIs/resources added/modified by the extension. This documentation is in `rst` format and is used to generate the OpenStack Networking API reference. The API reference lives under the `api-ref/source/v2` directory of the `neutron-lib` project repository.
- An extension descriptor class that must be defined in an extension directory for `neutron` or other sub-project that supports extensions. This concrete class provides the extensions metadata to the API server. These extension classes reside outside of `neutron-lib`, but leverage the base classes from `neutron_lib.api.extensions`. For more details see the section below on using neutron-libs extension classes.
- The API extension plugin implementation itself. This is the code that implements the extensions behavior and should carry out the operations defined by the extension. This code resides under its respective project repository, not in `neutron-lib`. For more details see the neutron api extension dev-ref.

Using neutron-libs base extension classes

The `neutron_lib.api.extensions` module provides a set of base extension descriptor classes consumers can use to define their extension descriptor(s). For those extensions that have an API definition in `neutron_lib.api.definitions`, the `APIExtensionDescriptor` class can be used. For example:

```
from neutron_lib.api.definitions import provider_net
from neutron_lib.api import extensions
```

(continues on next page)

(continued from previous page)

```
class Providernet(extensions.APIExtensionDescriptor):
    api_definition = provider_net
    # nothing else needed if default behavior is acceptable
```

For extensions that do not yet have a definition in `neutron_lib.api.definitions`, they can continue to use the `ExtensionDescriptor` as has been done historically.

3.6.3 API Converters

Definitions for REST API attributes, can include conversion methods to help normalize user input or transform the input into a form that can be used.

Defining A Converter Method

By convention, the name should start with `convert_to_`, and will take a single argument for the data to be converted. The method should return the converted data (which, if the input is None, and no conversion is performed, the implicit None returned by the method may be used). If the conversion is impossible, an InvalidInput exception should be raised, indicating what is wrong. For example, here is one that converts a variety of user inputs to a boolean value.

```
def convert_to_boolean(data):
     if isinstance(data, str):
         val = data.lower()
         if val == "true" or val == "1":
             return True
         if val == "false" or val == "0":
             return False
     elif isinstance(data, bool):
         return data
     elif isinstance(data, int):
         if data == 0:
             return False
         elif data == 1:
             return True
     msg = _("'%s' cannot be converted to boolean") % data
     raise n_exc.InvalidInput(error_message=msg)
```

Using Validators

In client code, the conversion can be used in a REST API definition, by specifying the name of the method as a value for the convert_to key on an attribute. For example:

```
'admin_state_up': {'allow_post': True, 'allow_put': True,
                   'default': True,
                   'convert_to': conversions.convert_to_boolean,
                   'is_visible': True},
```

Here, the admin_state_up is a boolean, so the converter is used to take users (string) input and transform it to a boolean.

Test The Validator

Do the right thing, and make sure youve created a unit test for any converter that you add to verify that it works as expected.

IPv6 canonical address formatter

There are several ways to display an IPv6 address, which can lead to a lot of confusion for users, engineers and operators alike. To reduce the impact of the multifaceted style of writing an IPv6 address, it is proposed that the IPv6 address in Neutron should be saved in the canonical format.

If a user passes an IPv6 address, it will be saved in the canonical format.

The full document is found at : http://tools.ietf.org/html/rfc5952

3.6.4 API Validators

For the REST API, attributes may have custom validators defined. Each validator will have a method to perform the validation, and a type definition string, so that the validator can be referenced.

Defining A Validator Method

The validation method will have a positional argument for the data to be validated, and may have additional (optional) keyword arguments that can be used during validation. The method must handle any exceptions and either return None (success) or a i18n string indicating the validation failure message. By convention, the method name is prefixed with `validate_` and then includes the data type. For example:

```
def validate_uuid(data, valid_values=None):
    if not uuidutils.is_uuid_like(data):
        msg = _("'%s' is not a valid UUID") % data
        LOG.debug(msg)
        return msg
```

There is a validation dictionary that maps the method to a validation type that can be referred to in REST API definitions. An entry in the dictionary would look like the following:

```
'type:uuid': validate_uuid,
```

Using Validators

In client code, the validator can be used in a REST API by using the dictionary key for the validator. For example:

```
NETWORKS: {
    'id': {'allow_post': False, 'allow_put': False,
           'validate': {'type:uuid': None},
           'is_visible': True,
           'primary_key': True},
    'name': {'allow_post': True, 'allow_put': True,
```

(continues on next page)

(continued from previous page)

```
                'validate': {'type:string': NAME_MAX_LEN},
                'default': '', 'is_visible': True}}
```

Here, the networks resource has an id attribute with a UUID validator, as seen by the validate key containing a dictionary with a key of type:uuid.

Any addition arguments for the validator can be specified as values for the dictionary entry (None in this case, NAME_MAX_LEN in the name attribute that uses a string validator). In a IP version attribute, one could have a validator defined as follows:

```
'ip_version': {'allow_post': True, 'allow_put': False,
               'convert_to': conversions.convert_to_int,
               'validate': {'type:values': [4, 6]},
               'is_visible': True}}
```

Here, the validate_values() method will take the list of values as the allowable values that can be specified for this attribute.

Test The Validator

Do the right thing, and make sure youve created a unit test for any validator that you add to verify that it works as expected, even for simple validators.

3.6.5 Neutron Callback System

In Neutron, core and service components may need to cooperate during the execution of certain operations, or they may need to react upon the occurrence of certain events. For instance, when a Neutron resource is associated to multiple services, the components in charge of these services may need to play an active role in determining what the right state of the resource needs to be.

The cooperation may be achieved by making each object aware of each other, but this leads to tight coupling, or alternatively it can be achieved by using a callback-based system, where the same objects are allowed to cooperate in a loose manner.

This is particularly important since the spin off of the advanced services like VPN and Firewall, where each services codebase lives independently from the core and from one another. This means that the tight coupling is no longer a practical solution for object cooperation. In addition to this, if more services are developed independently, there is no viable integration between them and the Neutron core. A callback system, and its registry, tries to address these issues.

In object-oriented software systems, method invocation is also known as message passing: an object passes a message to another object, and it may or may not expect a message back. This point-to-point interaction can take place between the parties directly involved in the communication, or it can happen via an intermediary. The intermediary is then in charge of keeping track of who is interested in the messages and in delivering the messages forth and back, when required. As mentioned earlier, the use of an intermediary has the benefit of decoupling the parties involved in the communications, as now they only need to know about the intermediary; the other benefit is that the use of an intermediary opens up the possibility of multiple party communication: more than one object can express interest in receiving the same message, and the same message can be delivered to more than one object. To this aim, the intermediary is the entity that exists throughout the system lifecycle, as it needs to be able to track whose interest is associated to what message.

In a design for a system that enables callback-based communication, the following aspects need to be taken into account:

- how to become consumer of messages (i.e. how to be on the receiving end of the message);
- how to become producer of messages (i.e. how to be on the sending end of the message);
- how to consume/produce messages selectively;

Translate and narrow this down to Neutron needs, and this means the design of a callback system where messages are about lifecycle events (e.g. before creation, before deletion, etc.) of Neutron resources (e.g. networks, routers, ports, etc.), where the various parties can express interest in knowing when these events for a specific resources take place.

Rather than keeping the conversation abstract, let us delve into some examples, that would help understand better some of the principles behind the provided mechanism.

Event payloads

The use of `**kwargs` for callback event payloads is deprecated (slated to be removed in Queens) in favor of standardized event payload objects as described herein.

The event payloads are defined in `neutron_lib.callbacks.events` and define a set of payload objects based on consumption pattern. The following event objects are defined today:

- `EventPayload`: Base object for all other payloads and define the common set of attributes used by events. The `EventPayload` can also be used directly for basic payloads that dont need to transport additional values.
- `DBEventPayload`: Payloads pertaining to database callbacks. These objects capture both the pre and post state (among other things) for database changes.
- `APIEventPayload`: Payloads pertaining to API callbacks. These objects capture details relating to an API event; such as the method name and API action.

Each event object is described in greater detail in its own subsection below.

For backwards compatibility the callback registry and manager still provide the `notify` method for passing `**kwargs`, but also provide the `publish` method for passing an event object.

Event objects: EventPayload

The `EventPayload` object is the parent class of all other payload objects and defines the common set of attributes applicable to most events. For example, the `EventPayload` contains the `context`, `request_body`, etc. In addition, a `metadata` attribute is available to transport event data thats not yet standardized. While the `metadata` attribute is there for use, it should only be used in special cases like phasing in new payload attributes.

Payload objects also transport resource state via the `states` attribute. This collection of resource objects tracks the state changes for the respective resource related to the event. For example database changes might have a pre and post updated resource thats used as `states`. Tracking states allows consumers to inspect the various changes in the resource and take action as needed; for example checking the pre and post object to determine the delta. State object types are event specific; API events may use python `dicts` as state objects whereas database events use resource/OVO model objects.

Note that states as well as any other event payload attributes are not copied; subscribers obtain a direct reference to event payload objects (states, metadata, etc.) and should not be modified by subscribers.

Event objects: DBEventPayload

For datastore/database events, `DBEventPayload` can be used as the payload event object. In addition to the attributes inherited from `EventPayload`, database payloads also contain an additional `desired_state`. The desired state is intended for use with pre create/commit scenarios where the publisher has a resource object (yet to be persisted) thats used in the event payload.

These event objects are suitable for the standard before/after database events we have today as well as any that might arise in the future.

Example usage:

```
# BEFORE_CREATE:
DBEventPayload(context,
               request_body=params_of_create_request,
               resource_id=id_of_resource_if_avail,
               desired_state=db_resource_to_commit)

# AFTER_CREATE:
DBEventPayload(context,
               request_body=params_of_create_request,
               states=[my_new_copy_after_create],
               resource_id=id_of_resource)

# PRECOMMIT_CREATE:
DBEventPayload(context,
               request_body=params_of_create_request,
               resource_id=id_of_resource_if_avail,
               desired_state=db_resource_to_commit)

# BEFORE_DELETE:
DBEventPayload(context,
               states=[resource_to_delete],
               resource_id=id_of_resource)

# AFTER_DELETE:
DBEventPayload(context,
               states=[copy_of_deleted_resource],
               resource_id=id_of_resource)

# BEFORE_UPDATE:
DBEventPayload(context,
               request_body=body_of_update_request,
               states=[original_db_resource],
               resource_id=id_of_resource
               desired_state=updated_db_resource_to_commit)

# AFTER_UPDATE:
DBEventPayload(context,
               request_body=body_of_update_request,
               states=[original_db_resource, updated_db_resource],
               resource_id=id_of_resource)
```

Event objects: APIEventPayload

For API related callbacks, the `APIEventPayload` object can be used to transport callback payloads. For example, the REST API resource controller can use API events for pre/post operation callbacks.

In addition to transporting all the attributes of `EventPayload`, the `APIEventPayload` object also includes the `action`, `method_name` and `collection_name` payload attributes permitting API components to pass along API controller specifics.

Sample usage:

```
# BEFORE_RESPONSE for create:
APIEventPayload(context, notifier_method, action,
          request_body=req_body,
          states=[create_result],
          collection_name=self._collection_name)

# BEFORE_RESPONSE for delete:
APIEventPayload(context, notifier_method, action,
          states=[copy_of_deleted_resource],
          collection_name=self._collection_name)

# BEFORE_RESPONSE for update:
APIEventPayload(context, notifier_method, action,
          states=[original, updated],
          collection_name=self._collection_name)
```

Subscribing to events

Imagine that you have entity A, B, and C that have some common business over router creation. A wants to tell B and C that the router has been created and that they need to get on and do whatever they are supposed to do. In a callback-less world this would work like so:

```
# A is done creating the resource
# A gets hold of the references of B and C
# A calls B
# A calls C
B->my_random_method_for_knowing_about_router_created()
C->my_random_very_difficult_to_remember_method_about_router_created()
```

If B and/or C change, things become sour. In a callback-based world, things become a lot more uniform and straightforward:

```
# B and C ask I to be notified when A is done creating the resource
# Suppose D another entity want subscription with higher priority
# notification
# ...
# A is done creating the resource
# A gets hold of the reference to the intermediary I
# A calls I
I->notify()
```

Since B and C will have expressed interest in knowing about As business, and D also subscribed for router creation with higher priority, I will deliver the messages to D first and then to B and C in any order. If B, C and D change, A and I do not need to change.

In practical terms this scenario would be translated in the code below:

```
from neutron_lib.callbacks import events
from neutron_lib.callbacks import resources
from neutron_lib.callbacks import registry


def callback1(resource, event, trigger, payload):
    print('Callback1 called by trigger: ', trigger)
    print('payload: ', payload)

def callback2(resource, event, trigger, payload):
    print('Callback2 called by trigger: ', trigger)
    print('payload: ', payload)

def callbackhighproirity(resource, event, trigger, payload):
    print("Prepared data for entities")

# A is using event in case for some callback or internal operations
registry.subscribe(callbackhighpriority, resources.ROUTER,
                   events.BEFORE_CREATE, priority=0)

# B and C express interest with I
registry.subscribe(callback1, resources.ROUTER, events.BEFORE_CREATE)
registry.subscribe(callback2, resources.ROUTER, events.BEFORE_CREATE)
print('Subscribed')


# A notifies
def do_notify():
    registry.publish(resources.ROUTER, events.BEFORE_CREATE,
                     do_notify, events.EventPayload(None))


print('Notifying...')
do_notify()
```

The output is:

```
> Subscribed
> Notifying...
> callbackhighpriority called by trigger: <function do_notify at␣
↪0x7f2a5d663410>
> payload: <neutron_lib._callbacks.events.EventPayload object at␣
↪0x7ff9ed253510>
> Callback2 called by trigger:  <function do_notify at 0x7f2a5d663410>
> payload: <neutron_lib._callbacks.events.EventPayload object at␣
↪0x7ff9ed253510>
> Callback1 called by trigger:  <function do_notify at 0x7f2a5d663410>
> payload: <neutron_lib._callbacks.events.EventPayload object at␣
↪0x7ff9ed253510>
```

Thanks to the intermediary existence throughout the life of the system, A, B, C and D are flexible to evolve their internals, dynamics, and lifecycles.

Since different entities can subscribe to same events of a resource, the callback priority mechanism is in place to guarantee the order of execution for callbacks, entities have to subscribe events with a priority number of Integer type, lower the priority number is higher would be priority of callback. The following

adds more details:

- Priorities for callbacks should be coded in *neutron_lib/callbacks/priority_group.py*
- If no priority is assigned during subscription then a default value will be used.
- For callbacks having same priority, the execution order will be arbitary.

Subscribing and aborting events

Interestingly in Neutron, certain events may need to be forbidden from happening due to the nature of the resources involved. To this aim, the callback-based mechanism has been designed to support a use case where, when callbacks subscribe to specific events, the action that results from it, may lead to the propagation of a message back to the sender, so that it itself can be alerted and stop the execution of the activity that led to the message dispatch in the first place.

The typical example is where a resource, like a router, is used by one or more high-level service(s), like a VPN or a Firewall, and actions like interface removal or router destruction cannot not take place, because the resource is shared.

To address this scenario, special events are introduced, BEFORE_* events, to which callbacks can subscribe and have the opportunity to abort, by raising an exception when notified.

Since multiple callbacks may express an interest in the same event for a particular resource, and since callbacks are executed independently from one another, this may lead to situations where notifications that occurred before the exception must be aborted. To this aim, when an exception occurs during the notification process, an abort_* event is propagated immediately after. It is up to the callback developer to determine whether subscribing to an abort notification is required in order to revert the actions performed during the initial execution of the callback (when the BEFORE_* event was fired). Exceptions caused by callbacks registered to abort events are ignored. The snippet below shows this in action:

```
from neutron_lib.callbacks import events
from neutron_lib.callbacks import exceptions
from neutron_lib.callbacks import resources
from neutron_lib.callbacks import registry


def callback1(resource, event, trigger, payload=None):
    raise Exception('I am failing!')

def callback2(resource, event, trigger, payload=None):
    print('Callback2 called by %s on event  %s' % (trigger, event))


registry.subscribe(callback1, resources.ROUTER, events.BEFORE_CREATE)
registry.subscribe(callback2, resources.ROUTER, events.BEFORE_CREATE)
registry.subscribe(callback2, resources.ROUTER, events.ABORT_CREATE)
print('Subscribed')


def do_notify():
    registry.publish(resources.ROUTER, events.BEFORE_CREATE, do_notify)

print('Notifying...')
try:
    do_notify()
```

(continues on next page)

(continued from previous page)

```
except exceptions.CallbackFailure as e:
    print("Error: %s" % e)
```

The output is:

```
> Subscribed
> Notifying...
> Callback2 called by <function do_notify at 0x7f3194c7f410> on event␣
↪before_create
> Callback2 called by <function do_notify at 0x7f3194c7f410> on event␣
↪abort_create
> Error:  Callback __main__.callback1 failed with "I am failing!"
```

In this case, upon the notification of the BEFORE_CREATE event, Callback1 triggers an exception that can be used to stop the action from taking place in do_notify(). On the other end, Callback2 will be executing twice, once for dealing with the BEFORE_CREATE event, and once to undo the actions during the ABORT_CREATE event. It is worth noting that it is not mandatory to have the same callback register to both BEFORE_* and the respective ABORT_* event; as a matter of fact, it is best to make use of different callbacks to keep the two logic separate.

Unsubscribing to events

There are a few options to unsubscribe registered callbacks:

- clear(): it unsubscribes all subscribed callbacks: this can be useful especially when winding down the system, and notifications shall no longer be triggered.
- unsubscribe(): it selectively unsubscribes a callback for a specific resources event. Say callback C has subscribed to event A for resource R, any notification of event A for resource R will no longer be handed over to C, after the unsubscribe() invocation.
- unsubscribe_by_resource(): say that callback C has subscribed to event A, B, and C for resource R, any notification of events related to resource R will no longer be handed over to C, after the unsubscribe_by_resource() invocation.
- unsubscribe_all(): say that callback C has subscribed to events A, B for resource R1, and events C, D for resource R2, any notification of events pertaining resources R1 and R2 will no longer be handed over to C, after the unsubscribe_all() invocation.

The snippet below shows these concepts in action:

```
from neutron_lib.callbacks import events
from neutron_lib.callbacks import exceptions
from neutron_lib.callbacks import resources
from neutron_lib.callbacks import registry


def callback1(resource, event, trigger, payload=None):
    print('Callback1 called by %s on event %s for resource %s' % (trigger,␣
↪event, resource))


def callback2(resource, event, trigger, payload=None):
    print('Callback2 called by %s on event %s for resource %s' % (trigger,␣
↪event, resource))
```

(continues on next page)

(continued from previous page)

```
registry.subscribe(callback1, resources.ROUTER, events.BEFORE_READ)
registry.subscribe(callback1, resources.ROUTER, events.BEFORE_CREATE)
registry.subscribe(callback1, resources.ROUTER, events.AFTER_DELETE)
registry.subscribe(callback1, resources.PORT, events.BEFORE_UPDATE)
registry.subscribe(callback2, resources.ROUTER_GATEWAY, events.BEFORE_
↪UPDATE)
print('Subscribed')


def do_notify():
    print('Notifying...')
    registry.publish(resources.ROUTER, events.BEFORE_READ, do_notify)
    registry.publish(resources.ROUTER, events.BEFORE_CREATE, do_notify)
    registry.publish(resources.ROUTER, events.AFTER_DELETE, do_notify)
    registry.publish(resources.PORT, events.BEFORE_UPDATE, do_notify)
    registry.publish(resources.ROUTER_GATEWAY, events.BEFORE_UPDATE, do_
↪notify)


do_notify()
registry.unsubscribe(callback1, resources.ROUTER, events.BEFORE_READ)
do_notify()
registry.unsubscribe_by_resource(callback1, resources.PORT)
do_notify()
registry.unsubscribe_all(callback1)
do_notify()
registry.clear()
do_notify()
```

The output is:

```
Subscribed
Notifying...
Callback1 called by <function do_notify at 0x7f062c8f67d0> on event before_
↪read for resource router
Callback1 called by <function do_notify at 0x7f062c8f67d0> on event before_
↪create for resource router
Callback1 called by <function do_notify at 0x7f062c8f67d0> on event after_
↪delete for resource router
Callback1 called by <function do_notify at 0x7f062c8f67d0> on event before_
↪update for resource port
Callback2 called by <function do_notify at 0x7f062c8f67d0> on event before_
↪update for resource router_gateway
Notifying...
Callback1 called by <function do_notify at 0x7f062c8f67d0> on event before_
↪create for resource router
Callback1 called by <function do_notify at 0x7f062c8f67d0> on event after_
↪delete for resource router
Callback1 called by <function do_notify at 0x7f062c8f67d0> on event before_
↪update for resource port
Callback2 called by <function do_notify at 0x7f062c8f67d0> on event before_
↪update for resource router_gateway
Notifying...
Callback1 called by <function do_notify at 0x7f062c8f67d0> on event before_
↪create for resource router
```

(continues on next page)

(continued from previous page)

```
Callback1 called by <function do_notify at 0x7f062c8f67d0> on event after_
↪delete for resource router
Callback2 called by <function do_notify at 0x7f062c8f67d0> on event before_
↪update for resource router_gateway
Notifying...
Callback2 called by <function do_notify at 0x7f062c8f67d0> on event before_
↪update for resource router_gateway
Notifying...
```

Subscribing events using registry decorator

Now neutron-lib supports using registry decorators to subscribe events. There are two decorators `has_registry_receivers`, which sets up the class `__new__` method to subscribe the bound method in the callback registry after object instantiation. `receives` use to decorate callback method which must defines the resource and events. Any class use `receives` must be decorated with `has_registry_receivers`.

Testing with callbacks

A python fixture is provided for implementations that need to unit test and mock the callback registry. This can be used for example, when your code publishes callback events that you need to verify. Consumers can use `neutron_lib.tests.unit.callbacks.base.CallbackRegistryFixture` in their unit test classes with the `useFixture()` method passing along a `CallbackRegistryFixture` instance. If mocking of the actual singleton callback manager is necessary, consumers can pass a value to with the `callback_manager` kwarg. For example:

```
def setUp(self):
    super(MyTestClass, self).setUp()
    self.registry_fixture = callback_base.CallbackRegistryFixture()
    self.useFixture(self.registry_fixture)
    # each test now uses an isolated callback manager
```

FAQ

Can I use the callbacks registry to subscribe and notify non-core resources and events?

> Short answer is yes. The callbacks module defines literals for what are considered core Neutron resources and events. However, the ability to subscribe/notify is not limited to these as you can use your own defined resources and/or events. Just make sure you use string literals, as typos are common, and the registry does not provide any runtime validation. Therefore, make sure you test your code!

What is the relationship between Callbacks and Taskflow?

> There is no overlap between Callbacks and Taskflow or mutual exclusion; as matter of fact they can be combined; You could have a callback that goes on and trigger a taskflow. It is a nice way of separating implementation from abstraction, because you can keep the callback in place and change Taskflow with something else.

Is there any ordering guarantee during notifications?

Depends, if the prorities are defined or passed during subscription, then yes callbacks will be executed in order, meaning the callback having the lowest integer value for priority will be executed first and so on. When priorities are not explicitly defined during subscription, all the callbacks will have default priority and will be executed in an arbitary order.

How is the notifying object expected to interact with the subscribing objects?

The `notify` method implements a one-way communication paradigm: the notifier sends a message without expecting a response back (in other words it fires and forget). However, due to the nature of Python, the payload can be mutated by the subscribing objects, and this can lead to unexpected behavior of your code, if you assume that this is the intentional design. Bear in mind, that passing-by-value using deepcopy was not chosen for efficiency reasons. Having said that, if you intend for the notifier object to expect a response, then the notifier itself would need to act as a subscriber.

Is the registry thread-safe?

Short answer is no: it is not safe to make mutations while callbacks are being called (more details as to why can be found line 937 of dictobject). A mutation could happen if a subscribe/unsubscribe operation interleaves with the execution of the notify loop. Albeit there is a possibility that things may end up in a bad state, the registry works correctly under the assumption that subscriptions happen at the very beginning of the life of the process and that the unsubscriptions (if any) take place at the very end. In this case, chances that things do go badly may be pretty slim. Making the registry thread-safe will be considered as a future improvement.

What kind of function can be a callback?

Anything you fancy: lambdas, closures, class, object or module methods. For instance:

```
from neutron_lib.callbacks import events
from neutron_lib.callbacks import resources
from neutron_lib.callbacks import registry


def callback1(resource, event, trigger, payload):
    print('module callback')


class MyCallback(object):

    def callback2(self, resource, event, trigger, payload):
        print('object callback')

    @classmethod
    def callback3(cls, resource, event, trigger, payload):
        print('class callback')


c = MyCallback()
registry.subscribe(callback1, resources.ROUTER, events.BEFORE_CREATE)
registry.subscribe(c.callback2, resources.ROUTER, events.BEFORE_CREATE)
registry.subscribe(MyCallback.callback3, resources.ROUTER, events.BEFORE_
↪CREATE)

def do_notify():
    def nested_subscribe(resource, event, trigger, payload):
```

(continues on next page)

(continued from previous page)

```
        print('nested callback')

    registry.subscribe(nested_subscribe, resources.ROUTER, events.BEFORE_
↪CREATE)

    registry.publish(resources.ROUTER, events.BEFORE_CREATE,
                     do_notify, events.EventPayload(None))


print('Notifying...')
do_notify()
```

And the output is going to be:

```
Notifying...
module callback
object callback
class callback
nested callback
```

3.6.6 DB Model Query

The implementation in `neutron_lib.db.model_query` is intended to be used as a stepping stone for existing consumers using standard database models/tables. Moving forward new database implementations should all use neutrons Versioned Object approach, while existing model based implementations should begin migrating to Versioned Objects.

Registering Hooks

The `neutron_lib.db.model_query.register_hook` function allows hooks to be registered for invocation during a respective database query.

Each hook has three components:

- query: used to build the query expression
- filter: used to build the filter expression
- result_filters: used for final filtering on the query result

Query hooks take as input the query being built and return a transformed query expression. For example:

```
def mymodel_query_hook(context, original_model, query):
    augmented_query = ...
    return augmented_query
```

Filter hooks take as input the filter expression being built and return a transformed filter expression. For example:

```
def mymodel_filter_hook(context, original_model, filters):
    refined_filters = ...
    return refined_filters
```

Result filter hooks take as input the query expression and the filter expression, and return a final transformed query expression. For example:

```
def mymodel_result_filter_hook(query, filters):
    final_filters = ...
    return query.filter(final_filters)
```

3.6.7 Neutron RPC API Layer

Neutron uses the oslo.messaging library to provide an internal communication channel between Neutron services. This communication is typically done via AMQP, but those details are mostly hidden by the use of oslo.messaging and it could be some other protocol in the future.

RPC APIs are defined in Neutron in two parts: client side and server side.

Client Side

Here is an example of an rpc client definition:

```
import oslo_messaging

from neutron_lib import rpc as n_rpc


class ClientAPI(object):
    """Client side RPC interface definition.

    API version history:
        1.0 - Initial version
        1.1 - Added my_remote_method_2
    """

    def __init__(self, topic):
        target = oslo_messaging.Target(topic=topic, version='1.0')
        self.client = n_rpc.get_client(target)

    def my_remote_method(self, context, arg1, arg2):
        cctxt = self.client.prepare()
        return cctxt.call(context, 'my_remote_method', arg1=arg1,␣
↪arg2=arg2)

    def my_remote_method_2(self, context, arg1):
        cctxt = self.client.prepare(version='1.1')
        return cctxt.call(context, 'my_remote_method_2', arg1=arg1)
```

This class defines the client side interface for an rpc API. The interface has 2 methods. The first method existed in version 1.0 of the interface. The second method was added in version 1.1. When the newer method is called, it specifies that the remote side must implement at least version 1.1 to handle this request.

Server Side

The server side of an rpc interface looks like this:

```
import oslo_messaging


class ServerAPI(object):

    target = oslo_messaging.Target(version='1.1')

    def my_remote_method(self, context, arg1, arg2):
        return 'foo'

    def my_remote_method_2(self, context, arg1):
        return 'bar'
```

This class implements the server side of the interface. The oslo_messaging.Target() defined says that this class currently implements version 1.1 of the interface.

Versioning

Note that changes to rpc interfaces must always be done in a backwards compatible way. The server side should always be able to handle older clients (within the same major version series, such as 1.X).

It is possible to bump the major version number and drop some code only needed for backwards compatibility. For more information about how to do that, see https://wiki.openstack.org/wiki/RpcMajorVersionUpdates.

Example Change

As an example minor API change, lets assume we want to add a new parameter to my_remote_method_2. First, we add the argument on the server side. To be backwards compatible, the new argument must have a default value set so that the interface will still work even if the argument is not supplied. Also, the interfaces minor version number must be incremented. So, the new server side code would look like this:

```
import oslo_messaging


class ServerAPI(object):

    target = oslo_messaging.Target(version='1.2')

    def my_remote_method(self, context, arg1, arg2):
        return 'foo'

    def my_remote_method_2(self, context, arg1, arg2=None):
        if not arg2:
            # Deal with the fact that arg2 was not specified if needed.
        return 'bar'
```

We can now update the client side to pass the new argument. The client must also specify that version 1.2 is required for this method call to be successful. The updated client side would look like this:

```
import oslo_messaging

from neutron.common import rpc as n_rpc


class ClientAPI(object):
    """Client side RPC interface definition.

    API version history:
        1.0 - Initial version
        1.1 - Added my_remote_method_2
        1.2 - Added arg2 to my_remote_method_2
    """

    def __init__(self, topic):
        target = oslo_messaging.Target(topic=topic, version='1.0')
        self.client = n_rpc.get_client(target)

    def my_remote_method(self, context, arg1, arg2):
        cctxt = self.client.prepare()
        return cctxt.call(context, 'my_remote_method', arg1=arg1,␣
↪arg2=arg2)

    def my_remote_method_2(self, context, arg1, arg2):
        cctxt = self.client.prepare(version='1.2')
        return cctxt.call(context, 'my_remote_method_2',
                          arg1=arg1, arg2=arg2)
```

More Info

For more information, see the oslo.messaging documentation: https://docs.openstack.org/oslo.messaging/latest/.