QA Specs Documentation

Release 0.0.1.dev221

OpenStack QA Team

Apr 25, 2023

CONTENTS

CHAPTER

ONE

SPECIFICATIONS

1.1 Tempest

1.1.1 Fuzzy test framework

https://blueprints.launchpad.net/tempest/+spec/fuzzy-test

The negative testing framework tests single aspects of an API server in an automatic manner based on json schemas. Using this functionality fuzzy tests can be created with the same process but with a different focus.

Problem description

Tempest does not have any coverage of security aspects. Using such a framework to detect security vulnerabilities will be an important new testing area for Tempest.

Proposed change

Focus of this framework is vulnerabilities identification and denial of service (DoS) attacks. It can use the api schema definitions as input to produce flawed requests.

Denial of service

DoS patterns are easy to validate and are considered as first step. A certain service produces a portion of flawed requests together with valid requests like authentication. To validate if a DoS attack was successful a set of usual Tempest API test can be used for this purpose. To produce the needed load the stress test framework of Tempest can be used to produce a higher load rate.

Vulnerabilities identification

Identification of security issues can be very complex and automatic detection can be only done very limited. To identify issues that may are vulnerabilities the following data needs to be analyzed:

- Result of a request: Success codes or internal server errors are potential threats that need be analyzed and logged by the framework.
- System availability: A check if all the OpenStack components are available be used as validation.
- Request logging: Tempest rest client loges all requests. This is needed to identify request or scenarios that causes a threat.

Data generation

The data generation should support different sources and this could be a possible interface to 3PP fuzzy testing products. With the multibackend functionality of the negative testing framework (see https://review.openstack.org/#/c/73982/) different test generators can be used. These generator must stick to the interface define in the base class (tempest.common.negative.base).

Alternatives

Use third party product for fuzzy test generation and dont integrate it in Tempest.

Implementation

Assignee(s)

Primary assignees:
Marc Koderer (mkoderer)

Milestones

Target Milestone for completion:
Juno-3

Work Items

Will be tracked in:
https://etherpad.openstack.org/p/bp_fuzzy_test

Dependencies

None.

```
This work is licensed under a Creative Commons Attribution 3.0 Unported
License.

http://creativecommons.org/licenses/by/3.0/legalcode
```

1.1.2 Simplify credentials management

https://blueprints.launchpad.net/tempest/+spec/simplify-credentials-management

Refactor the way credentials/client managers are obtained by the test classes so that it is clear which class attributes reference the client managers with specific credentials.

Problem description

Credentials are allocated by defining an array that enumerates the needed credentials for a given test class. For instance:

```
credentials = [['operator', CONF.object_storage.operator_role],
               ['operator_alt', CONF.object_storage.operator_role]]
```

When the `setup_credentials` class method of the base test class is called, the client managers associated with credentials are mapped to class attributes with the prefix `os_roles`. Credentials can also be allocated in this manner:

```
credentials = ['primary', 'alt', 'admin']
```

In this case the client managers are aliased to three class attributes, for instance `os`, `manager`, `os_primary` are all set to the client manager using the `primary` credentials. This can be confusing for someone trying to understand a test case, because it is not intuitive for the setting of a class variable to result in attributes being set and what the names of those attributes are.

Proposed change

The proposed change is to explicitly assign the attribute values in the given classs `setup_credentials` class method; for instance:

```
cls.os_roles_operator = cls.get_client_manager(
    roles=[CONF.object_storage.operator_role], force_new=True)
cls.os_roles_operator_alt = cls.get_client_manager(
    roles=[CONF.object_storage.operator_role], force_new=True)
```

For classes that use the `primary`, `alt` and/or `admin` credentials, the logic would look like this:

```
cls.os_primary = cls.get_client_manager(credential_type='primary')
```

All aliasing would be removed.

In either case the credential de-allocation logic can remain the way it is.

Implementation

Assignee(s)

John Warren <jswarren@us.ibm.com>

Milestones

Work Items

- tempest/scenario/test_server_multinode.py and subclasses
- tempest/scenario/test_security_groups_basic_ops.py and subclasses
- tempest/scenario/test_aggregates_basic_ops.py and subclasses
- tempest/scenario/manager.py and subclasses
- tempest/api/database/base.py and subclasses
- tempest/api/compute/base.py and subclasses
- tempest/api/compute/test_authorization.py and subclasses
- tempest/api/compute/servers/test_servers_negative.py and subclasses
- tempest/api/telemetry/base.py and subclasses
- tempest/api/baremetal/admin/base.py and subclasses
- tempest/api/object_storage/base.py and subclasses
- tempest/api/object_storage/test_object_services.py and subclasses
- tempest/api/object_storage/test_account_services.py and subclasses
- tempest/api/object_storage/test_account_quotas.py and subclasses
- tempest/api/object_storage/test_container_acl_negative.py and subclasses
- tempest/api/object_storage/test_account_services_negative.py and subclasses
- tempest/api/object_storage/test_container_sync.py and subclasses
- tempest/api/object_storage/test_container_acl.py and subclasses
- tempest/api/object_storage/test_account_quotas_negative.py and subclasses
- tempest/api/data_processing/base.py and subclasses
- tempest/api/network/admin/test_floating_ips_admin_actions.py and subclasses
- tempest/api/network/base.py and subclasses
- tempest/api/volume/base.py and subclasses
- tempest/api/volume/test_volume_transfers.py and subclasses

- tempest/api/identity/base.py and subclasses
- tempest/api/identity/v3/test_projects.py and subclasses
- tempest/api/identity/v2/test_tenants.py and subclasses
- tempest/api/image/base.py and subclasses
- tempest/api/orchestration/base.py and subclasses

1.2 DevStack

1.2.1 Inspect counters for data collection and gating purposes

Problem description

OpenStack projects vary on their impact to the underlying infrastructure that they rely on greatly. This is hard to measure without going to a full scale deployment, but we should be able to measure the impact by inspecting counters already maintained by the system.

Proposed change

- Create a new OpenStack QA Tools repository to house small tools written in python for purposes such as this.
  - Create a tool, *os-collect-counters*, which collects relevant counters from any backends it can reach using its own configuration and outputs a JSON mapping with all counters. Includes ability to delta with a previous run to allow showing impact on the counters for a given time window.
    - Initial counters will be at a minimum a set of MySQL counters (such as Innodb_bytes_written) and published messages from the RabbitMQ management interface, summarized by scope that can be inferred from each queue name.
- Leverage existing subunit/statsd/graphite infrastructure to record results of several tests in the devstack gate.
  - For each run, the JSON from *os-collect-counters* will be added as an attachment to the subunit stream.
  - The counters in the attachment will be fed into statsd/graphite to allow establishing trends.
    - This will be facilitated by adding attachment storage plugins to subunit2sql. The plugin used for OpenStack gate jobs will be specific to OpenStacks infrastructure and look for the specifically named attachment to push into statsd/graphite.
- Monitor counters for stable indicators and identify the best predictors of problems.
  - Once stable counters are identified, create an upper bounds for these counters to help prevent new changes in the system from accidentally introducing an inordinate amount of cost into the tested code paths.

    Since there are daunting social issues around failing gate tests on global collisions, warnings and bugs about said warnings are likely the only reasonable outcome we can achieve. It will take a considerable amount of community agreement to make these limits hard.

Implementation

A new python repo, os-performance-tools, has already been created, and will be maintained for the purposes of extracting and pushing counters into statsd/graphite. This will include a subunit2sql attachments plugin and code to output the counters as a subunit attachment.

Assignee(s)

Primary assignee:

- Clint Byrum <clint@fewbar.com>

Milestones

Target Milestone for completion:
Mitaka-2

Work Items

- Create tools to emit counters from a running installation
- Modify devstack gate job output to include counters
- Add subunit2sql attachment plugin to subunit2sql workers to push counters to infra graphite
- Analyze data for stable counters and useful trends
- Add upper bounds check to devstack gate

Dependencies

References

1.3 Other

1.3.1 Placeholder

This is just a placeholder file to avoid sphinx errors. Please remove this file when you add a new rst file.

1.3.2 Whitebox Tempest Plugin

https://blueprints.launchpad.net/tempest/+spec/whitebox-tempest-plugin

Problem description

Tempest defines its scope as only what is accessible through the various REST APIs. Some cloud features cannot be properly tested when using only the REST API. For example, in the context of Nova using the libvirt driver, certain features can only be fully tested by examining the XML of the instances. Specifically, live-migrating an instance with dedicated CPUs can appear to succeed, while in actuality the live migration has caused the instance to no longer have dedicated CPUs. Only by looking at the instances XML can we validate that the dedicated CPU SLA has been respected.

Proposed change

The whitebox-tempest-plugin is a Tempest plugin whose scope explicitly requires peeking behind the curtain. In other words, if a feature or behavior can be fully tested using only a REST API, such a test does not belong in whitebox-tempest-plugin. On the other hand, if fully testing a feature or behavior requires accessing the control plane like a human operator or admin would, such a test belongs in whitebox-tempest plugin.

The plugin provides a framework for tests to look behind the curtain. It currently contains Tempest-style clients that can examine an instances XML, examine the database, read and write INI configuration options, and restart services. All of these are used by tests that are concentrated around Nova NFV features like CPU pinning and NUMA-aware live-migration.

While currently heavily centered on Nova and NFV, whitebox-tempest-plugin aims to be useful for testing any OpenStack project.

Alternatives

While various team-specific or project-specific Tempest plugins exist whose scope sometimes intersects with whitebox-tempest-plugin, the whitebox testing use case does not currently have an alternative community solution.

Projects

Whitebox-tempest-plugin is self-contained. It includes a devstack plugin to add whitebox-specific options to tempest.conf.

Implementation

Assignee(s)

The current whitebox-tempest-plugin team is composed of Joe Hakim Rahme (rahmu), Sean Mooney (sean-k-mooney) and Artom Lifshitz (notartom).

Milestones

Ussuri.

Work Items

The current roadmap consists of:

- Start using whitebox-tempest-plugin in the Nova NFV Zuul job.
- Whitebox supports undercloud/overcloud TripleO deployments, but there is no upstream CI coverage for this. Work is in progress to add a TripleO CI job. The new job will use a multinode standalone TripleO deployment, which is also in the process of being developed.

Dependencies

Whitebox requires the following Python modules:

- sshtunnel (needed to access the overcloud database, will be dropped once the TripleO CI job is done, as a standalone TripleO deployment does not have the undercloud/overcloud distinction).
- pymysql (needed to intereact with the database)

References

- Nova NFV Zuul job: https://review.opendev.org/#/c/679656/
- TripleO CI job: https://review.opendev.org/#/c/705113/

CHAPTER

TWO

IMPLEMENTED SPECIFICATIONS

2.1 Tempest

```
This work is licensed under a Creative Commons Attribution 3.0 Unported
License.

http://creativecommons.org/licenses/by/3.0/legalcode
```

2.1.1 Add Swift API Tests for Icehouse

https://blueprints.launchpad.net/tempest/+spec/add-icehouse-swift-tests

Add Swift API tests which are added in Icehouse release (version 1.13.1)

Problem description

Between Havana and Icehouse releases, some new features are added in Swift. However, Tempest currently has only subset of API tests of those features.

Proposed change

Add API tests for following new functions.

- New-style container synchronization
- Getting contents inline by TempURL
- POST request to delete multiple containers and objects in bulk
- PUT object with If-None-Match: * header

New file test_container_sync_middleware.py will be created to include tests of new container synchronization. Test cases for other two features are added in existing appropriate files.

In new container sync feature, realm and cluster names are used in X-Container-Sync-To header like //<realm_name>/<cluster_name>/<account>/<container> to specify where to synchronize objects as substitute for URL which is used in old-style container sync. Realm and cluster names are defined in Swifts container-sync-realms.conf, therefore it is also necessary to specify realm and cluster names in tempest.conf. Following two config values must be added:

```
[object-storage]
realm_name=<realm name>
cluster_name=<cluster name>
```

Implementation

Assignee(s)

Daisuke Morita <morita.daisuke@lab.ntt.co.jp>

Milestones

Target Milestone for completion:
Juno-3

Work Items

- Write test cases for Swifts new functions
- Add config values to run tests of new-style container sync

Working progress will be tracked in http://goo.gl/qRLgZe (Google Doc).

2.1.2 Add service tags to tests

https://blueprints.launchpad.net/tempest/+spec/add-service-tags

Add new tags to all tests that specify which services get exercised by the test.

Problem description

When running tempest there is no clear way to specify only run tests that hit a subset of services. For example, if you wanted to only run tests that used cinder for the purposes of verifying a new driver there isnt a method to easily filter the tests run so that only cinder tests are run. The only option is to manually construct a regex filter that executes the tests which you think hit cinders api.

Proposed change

To add a new decorator which will set a service attr for the test specified. The decorator will have a single parameter which will be a list of the services that the test exercises. The decorator will only except valid service functional names in the list, for example compute, volumes, etc. If an invalid service is passed into the decorator it will error out. The end result will be that if you run tempest with the functional name of a service as a regex filter you will only run the tests that touch the service directly or indirectly (ie through a proxy api, like novas images api) So for the example in the problem statement you would run:

```
testr run --parallel volumes
```

Or some variation of the command and only tests that uses cinder would be run.

The service decorator is only required if the service name is not in the path for the test. If a test exercises a service that contains the name in the path then its redundant to use the service decorator because the regex filter will already match. For the scenario tests since the is normally not a service name in the path service tags are required for each test. This will be enforced with a hacking rule.

A test that is properly decorated will look something like this:

```
@test.services('compute', 'volume', 'image', 'network')
def test_minimum_basic_scenario(self):
```

which indicates that test_minimum_basic_scenario uses the compute, volume, image, and networking APIs.

An additional feature of adding service tags is that by tagging the tests we know that the service is required to run the test. This means we can skip the test if the required service is set as not available in the config file. This means that an additional skip decorator or skip exception wont be needed if for tests where service tagging is applicable. However, for cases where service tags shouldnt be used, such as places where the patch already contains the name, the other skip methods will be required. (which they should already have)

The decorator will be put in tempest/test.py while all the test methods in any of the tempest test categories are subject to having the decorator applied to them. Of course thats assuming the previously mentioned usage conditions are met by the test in question.

Implementation

Assignee(s)

Matthew Treinish <mtreinish@kortar.org>

Milestones

Target Milestone for completion:
Juno-1

Work Items

- Add service decorator
- Add service tags to scenario tests
- Create Hacking Extension to force service tags in scenario tests
- Create Hacking Extension to ensure service tag isnt in module path
- Add service tags to applicable volume api tests
- Add service tags to applicable compute api tests
- Add service tags to applicable image api tests
- Add service tags to applicable identity api tests
- Add service tags to applicable network api tests
- Add service tags to applicable orchestration api tests
- Add service tags to applicable object api tests

2.1.3 Tempest support for API microversions testing

https://blueprints.launchpad.net/tempest/+spec/api-microversions-testing-support

Since Kilo, nova has implemented API microversions and the other components (Ironic, etc) also implemented it now. However, currently Tempest does not have any support for microversions and doesnt test it at all. This proposal is to add microversions testing support in Tempest.

Problem description

On microversions mechanism, each microversion change is very small. For example, version 2.2s change is:

```
Add 'type' to os-keypairs response
Change status code for os-keypairs create method from 200 to 201
Change status code for os-keypairs delete method from 202 to 204
```

In long term, a lot of microversions will be implemented because all API changes should be done with microversions. Now Ironic also implements this microversions and the other projects have a plan to implement it. So we need to implement consistent basic test way for these projects.

Proposed change

Test classes for each microversion

- Implement test classes for each microversion When adding a new microversion which changes an API, basically we need to implement a test class for the API. In addition, each test class contains its microversion range with class values like min_microversion and max_microversion. For example, we have added a new attribute type to os-keypairs response on novas microversion 2.2, and the corresponding test class will be:

```
class KeyPairsV22Test(base.BaseKeypairTest):
    min_microversion = '2.2'

    [..]
```

  as os-keypairs test class. In the above case, max_microversion is not contained. That means unlimited as max_microversion. If we change the API again with microversion 2.100 as an example, the test class will be:

```
class KeyPairsV22Test(base.BaseKeypairTest):
    min_microversion = '2.2'
    max_microversion = '2.99'

    [..]
```

  and we need to add a test like:

```
class KeyPairsV100Test(base.BaseKeypairTest):
    min_microversion = '2.100'

    [..]
```

- Add configuration options for specifying test target microversions We need to specify test target microversions because the supported microversions are different between OpenStack clouds. For operating multiple microversion tests in a single Tempest operation, configuration options should represent the range of test target microversions. New configuration options also are min_microversion and max_microversion, and the test classes will be selected like the following:

```
TestClass A: min_microversion = None,  max_microversion = 'latest'
TestClass B: min_microversion = None,  max_microversion = '2.2'
TestClass C: min_microversion = '2.3', max_microversion = 'latest'
TestClass D: min_microversion = '2.5', max_microversion = '2.10'
```

Configuration (min, max)	Test classes (Passed microversion)
None, None	A(Not passed), B(Not passed), C & D - Skipped
None, 2.3	A(Not passed), B(Not passed), C(2.3), D - Skipped
2.2, latest	A(2.2), B(2.2), C(2.3), D(2.5)
2.2, 2.3	A(2.2), B(2.2), C(2.3), D - Skipped
2.10, 2.10	A(2.10), B - Skipped, C(2.10), D(2.10)
None, latest	A(Not passed), B(Not passed), C(2.3), D(2.5)
latest, latest	A(latest), B - Skipped, C(latest), D - Skipped

So basically the configuration min_microversion value is passed on the microversion header. However if the selected class min_microversion is bigger, the class min_microversion is passed instead. If youd like to always pass the maximum micoversion then, you need to set the max_microversion and the min_microversion to be the same value, like the 5th example above.

The default configuration values should be (None, None) like 1st example for running on the existing clouds which dont support microversions. So we need to change the configuration values with openstack-dev/devstack and openstack-infra/project-config for operating microversion tests on the gate.

The microversion latest is a magic keyword as final example. When passing latest as the microversion to each component(Nova, etc.), the component takes the latest microversion action on the server side. Some microversions will be backwards incompatible and the latest action can break the gate test if Tempest doesnt support the microversion at the time. To avoid such situation, we should not specify latest on regular gate jobs. It is nice to specify it as experimental job to know we need to update Tempest for supporting the latest microversion.

These configuration options should be added for each project(Nova, Ironic, etc.) because the microversion numbers are different between projects.

JSON-Schema for each microversion (Nova specific)

- Define responses for each microversion Backwards compatible changes also need new microversions on Novas microversions and Tempest is verifying it by checking Nova API responses dont contain any extra attributes with JSON-Schema additionalProperties feature. So we need to define the responses for each microversions and Tempest needs to switch the response definition of JSON-Schema by the microversion. Now the responses are defined under tempest_lib/api_schema/response/compute/ of tempest-lib and the one of the base microversion v2.1 is defined under ./v2_1 . Each microversion is a little different from the previous one and it is necessary to define the difference under ./v2_2, ./v2_3, etc.
- Make service clients switch response definition for each microversion Service clients of Nova will switch the definition based on the microversion.

Tempest-lib migration plan

- Steps:
  1. Implement the microversion testing framework in Tempest. The framework includes skipping methods etc for microversion tests based on the provided configuration.
  2. Implement base framework for service clients to pass microversion to a request header in Tempest.
  3. Implement tests case for Nova microversion v2.2 as sample in Tempest. This includes schema and service client change also. We can test the microversion testing framework at this time, and it will be ready to migrate the framework to tempest-lib.
  4. Migrate the microversion testing framework to tempest-lib

External consumption

Once all frameworks are migrated to Tempest-lib, other projects can use the same for their microversion testing. Document needs to be updated how to consume the microversion testing framework with some example.

Projects

- openstack/tempest
- openstack/tempest-lib
- openstack-dev/devstack
- openstack-infra/project-config

Implementation

Assignee(s)

- Kenichi Ohmichi <ken-oomichi@wx.jp.nec.com>
- Ghanshyam Mann <ghanshyam.mann@nectechnologies.in>
- Yuiko Takada <yui-takada@tg.jp.nec.com>

Milestones

Target Milestone for completion:
Mitaka-1

Work Items

- Implement base test classe for microversions
- Pass a test target microversion to service clients
- Add a test class for a single microversion(as sample)
- Migrate tested microversion testing framework to Tempest-lib
- Consume those interface from Tempest-lib and remove from Tempest
- Change the configurations on openstack-infra/project-config for master

Dependencies

None

References

- https://review.openstack.org/#/c/242296/

2.1.4 API schema unification

https://blueprints.launchpad.net/tempest/+spec/api-schema-unification

API schemas are used for different purposes in Tempest. This blueprint tries to unify all existing ways.

Problem description

Tempest currently has two sources of schema definitions:

- The response validation framework (`tempest/api_schema`)
- The negative test framework, which automatically creates requests (`etc/schema`)

Differences in a nutshell:

- File type - `etc/schema` files are json based - `tempest/api_schema` files are python modules
- Data type - `etc/schema` contains Tempest related data (result code check base on generators) - `tempest/api_schema` contains data that can be imported from projects
- Content - `etc/schema` is used for request generation - `tempest/api_schema` is used for response validation

Proposed change

Move all schemas to tempest/api_schema and use .py files instead of .json files. This gives the possibility to use inheritance (or any other python magic) to reduce code duplication. Inside of the py files the data format will be dicts instead of json. This is due to the fact that all existing definitions are already defined as dicts.

The proposed folder structure:

```
tempest
|-> api_schema
|   |-> request # old content of ``etc/schema``
|   |   |-> compute
|   |   |   |-> v2
|   |   |   |-> v3
|   |-> response # old content of ``tempest/api_schema``
|   |   |-> compute
|   |   |   |-> v2
|   |   |   |-> v3
```

Next steps

Out of scope of this blueprint but next steps:

- Replace dicts with json definitons
- Having same/similar json style
- Using same load mechanism

Alternatives

To be discussed.

Implementation

Assignee(s)

Primary assignees:
Marc Koderer (mkoderer)

Milestones

Target Milestone for completion:
Juno-final

Work Items

1. Move all files to one location
2. Rewrite all .json files to .py files and adapt negative testing framework
3. Unify filenames to have consistence between /response and /request

Dependencies

None.

2.1.5 Branchless Tempest

https://blueprints.launchpad.net/tempest/+spec/branchless-tempest

Tempest historically has been treated like a core service, with a stable/foo release branch cut at every release of OpenStack. However, Tempest is largely black box testing for the OpenStack API. The API is represented by major versions, not by release tags, so Tempest should not need branches. This change would have a number of interesting cascading impacts, all of which should be positive for the health of the project.

Problem description

Our current release method for Tempest is to create a stable/foo branch at every release of the OpenStack core services. This has a number of side effects.

- tests are only added to master, and rarely backported. Completely valid stable/havana testing has been lost.
- behavior changes can happen between stable/havana and master because the version of the test run between them is different
- what version of Tempest should a CDing public cloud test against? (i.e. releases arent meaningful to lots of consumers)
- stable/havana ends up being used as a global flag for what features are supported in OpenStack. But thats not meaningful because most environments run with a specific list of features enabled, not just anything we shipped in stable/havana.

Tempest has organically grown up as a tool that works really well in our upstream gate, and medium well for people testing real deployments. One of the reasons for this difference is that we use the stable/foo Tempest branches to make assumptions about the relationship of services, and devstack, which set up Tempest. Breaking this convenience relationship will make us be much more dilligent in being explicit about Tempest as a tool designed to run under any setup environment.

Proposed change

Under the proposed branchless environment wed do the following:

- Not set a stable/icehouse Tempest branch
- stable/icehouse integrated services would be tested with Tempest master
- Tempest master would be gated on successful runs against:
  - integrated services on master branches
  - integrated services on stable/icehouse branches
  - (Note: this would double the # of jobs to be run on Tempest proposed changes)
- post release, devstack-gate would change to explicitly set not only the services that Tempest expects to tests, but also the extensions it expects to test on each branch

This has cascading implications, probably best described as scenarios (Ill use nova as the example service for all scenarios, mostly because Im more familiar with the extensions model, not because its being picked on):

Scenario 1: New Tests for new features

If a new feature is added to Nova in Juno, and a new test is added to Tempest for this feature, this test will need to be behind a feature flag (as the feature wasnt available in Icehouse).

- nova stable/icehouse - Test Skipped, feature not available
- nova master - Test run

This has the added benefit of making sure that a new Nova change is added in such a way that its somehow discoverable that its not there (i.e. behind an unloaded extension), and that Tempest has a knob for configuring or not configuring it.

Scenario 2: Bug fix on core project needing Tempest change

Previously a behavior change in nova master that needed a Tempest change could be changed via the tempest 2 step:

- propose change to nova, get a nova +2
- propose skip on Tempest, +Aed after nova +2 on change
- land nova change in master and stable/icehouse (if required)
- land changed test in Tempest

In this new model the change *will also* need to be backported to stable/icehouse simultaneously. Or we decide that this is behavior that should not be tested for, and drop the test entirely.

Scenario 3: New Tests for existing features

When adding new tests for existing features the new tests will need to simultaneously pass when tested against nova master and nova stable/icehouse. If we discover there is a difference of behavior between the two, well need to harmonize that behavior before landing the test. We end up with 3 options:

- fix nova master to reflect nova stable/icehouse behavior (nova regression)
- fix nova stable/icehouse to reflect nova master behavior (missing backport)
- decide to not land the Tempest test as it is attempting to verify behavior we dont consider stable

Scenario 4: Obsolete Tests

A feature, like Nova XML v2 is deprecated is icehouse, and (hypothetically) removed in juno. Howeve the XML v2 tests are in Tempest, and we still need to test icehouse releases.

This would be handled by a slow deprecation / feature flag:

- Tempest v2 XML feature flag added (default to false)
- Devstack-gate modified to set to true for stable/icehouse
- Tests are skipped in master, run in stable/icehouse
- Once stable/icehouse is no longer supported upstream (Feb 2015), tests are removed from Tempest entirely.

This will mean that Tempest will contain more legacy code, however thats a trade off we take with the benefits this provides.

Additional Implications

This will have some interesting additional fallout:

- stable/* branches wont be broken so often: with the massive number of tempest patches that will require a working stable/* gate to land, stable/* will get a lot more regular testing, and be in a working state more often
- API guaruntees will tighten up. Well not only be testing that the API does what we expect in Tempest, but also that it acts the same across multiple versions. The added friction to breaking that kind of behavior will slow down any future breaks there. This means a practical increase of API compatibility requirements.

Alternatives

The alternative is to continue with the current approach which uses stable/* branches in Tempest. However that has caused more unintentional coupling with devstack that wed like, and I believe that long term its the wrong approach for a test suite like Tempest.

Implementation

Assignee(s)

Primary assignee:
Sean Dague <sean@dague.net>

Can optionally can list additional ids if they intend on doing substantial implementation work on this blueprint.

Milestones

Target Milestone for completion:

- April 17th for main infrastructure
- Juno release cycle to handle various backports to devstack stable/icehouse to support all tempest feature flags

Work Items

The work items span projects

- devstack-gate generic branch override support (DONE)
- infrastructure jobs to gate Tempest on stable/icehouse (when available)
- devstack-gate changes to select not only services, but also features supported at each release
- devstack support for setting stable/icehouse features
- additional Tempest feature flags for optional features not yet addressed

Dependencies

Only those listed above

2.1.6 Branchless Tempest - Service Extensions

https://blueprints.launchpad.net/tempest/+spec/branchless-tempest-extensions

This is the follow on work for branchless tempest for new service extensions that are added over the course of a release.

Problem description

In moving to branchless Tempest were now running Tempest across multiple OpenStack code branches. Today thats icehouse and juno, however it will be icehouse, juno, kepler in the future.

What happens when a new extension is added to Nova in Juno, and tests in tempest are wanted for that part of the API? Today that test would fail because it could not pass icehouse.

Proposed change

The proposed change is to add another layer to the devstack-gate feature grid which specifies which extensions are supported at each release.

Today in the nova definition weve got

```
nova:
  base:
     services: [n-api, n-cond, n-cpu, n-crt, n-net, n-obj, n-sch]
```

This would imagine a world where the definition would look as follows

```
nova:
  base:
     services: [n-api, n-cond, n-cpu, n-crt, n-net, n-obj, n-sch]
  icehouse:
     compute-ext: [floating-ips, aggregates, ... ]
```

The non existence of an extensions list means assume all. It is also expected that youd be able to specify rm-compute-ext much like rm-services, so that you could do something as follows.

```
nova-cells:
  base:
    services: [n-cell]
    rm-compute-ext: [aggregates, hosts]
```

That would disable those nova extensions any time it was configured.

For this to function there needs to be changes in

- devstack-gate
  - to parse these additional stanzas and pass them down to devstack
- devstack
  - to take extension lists for projects and set the correct extensions up based on it
  - to compute the all case correctly for master (especially if we support the rm-compute-ext stanza)
  - to set the correct tempest fields for enabled features when these map to feature flags.

Alternatives

Nova API microversions might obviate the need here in the branch case, as wed be able to specify a specific test has a specific required version. However that wouldnt solve the cells case.

Implementation

Assignee(s)

Primary assignee:
None yet

Milestones

Not Known

Work Items

The work items span projects

- see above

Dependencies

Only those listed above

2.1.7 Centralized Tempest Workspace Management

https://blueprints.launchpad.net/tempest/+spec/centralized-workspaces

Create a consistent means for creation and management of Tempest workspaces.

Problem description

Currently there is no way to track workspaces in a consistent manner. This becomes problematic as the number of workspaces increases.

Proposed change

Create a `.tempest` file in the users home directory to be used as a source of truth for Tempest workspaces. Users can register new workspaces via the `tempest workspace register` command. New workspaces are automatically registered via `tempest init`. The workspace manager automatically unregisters any workspaces that no longer exist.

Action	Command
Register a workspace:	tempest workspace register name <name> path <path>
Rename a workspace:	tempest workspace update key <key> old-value <old> new-value <new>
List workspaces:	tempest workspace list

Example Usage

```
> cd ~/devstack
> tempest init --name devstack

> tempest workspace register --name staging --path /etc/staging

> tempest workspace list
+----------+----------------+
| Name     |    Location    |
+----------+--------------- +
| devstack | /root/devstack |
| staging  | /etc/staging   |
+----------+----------------+
```

Projects

- openstack/tempest

Implementation

Assignee(s)

- slowrie
- dwalleck

Milestones

Target Milestone for completion:

- Mitaka-2

Work Items

- Create argparse to handle new `workspace` command and subcommands
- Create tracking file and class to represent it
- Add code to list that unregisters workspaces when locations no longer exist

References

- https://etherpad.openstack.org/p/tempest-cli-improvements

2.1.8 Implement Cinder V2 API test

https://blueprints.launchpad.net/tempest/+spec/cinder-v2-api-tests

Implement Cinder v2 API test by sharing service client and test code with v1.

Problem description

Tempest doesnt have enough Cinder v2 api tests. We need to add more tests for it. Cinder v2 api only has some small updates, so v1 and v2 tests could share service client and test code. In this way, we dont need to maintain many duplictate test codes.

Proposed change

This blueprint proposes that Cinder v1 and v2 tests share service client and test code. It includes the following changes:

1. Create common service clients
2. Create common base test class
3. Make each v1 test class inherit the v2 test class
4. Change the Cinder API test directory structure

1. Create common service clients

v1 and v2 service client only have very little difference. Most codes should be same. So we could create a common service client and inherit it by v1 and v2.

- [v1 service client] -> [CommonServiceClient]
- [v2 service client] -> [CommonServiceClient]

2. Create common base test class

Now each Cinder API test class inherits like:

- [v1 test class] -> [BaseVolumeV1Test] -> [BaseVolumeTest]
- [v2 test class] -> [BaseVolumeV2Test] -> [BaseVolumeTest]

To share API test classes, we need to create common base test class instead of BaseVolumeV1Test/BaseVolumeV2Test. The class instance can switch its behavior based on some variable which represents an API version. After applying this common class to every API test classes, the existing BaseVolumeV1Test and BaseVolumeV2Test can be removed.

3. Make each v1 test class inherit the v2 test class

We need to change v2 test classes inheritances to the common base test class and change v1 test classes inheritances to v2 test class:

- [v1 test class] -> [v2 test class] -> [CommonVolumeTest]

In v1 test class, the variable which represents an API version should be 1, and the variable should be 2 in v2 test class:

```
class SomeApiV2Test(CommonVolumeTest):
  _api_version = 2
  [..]

class SomeApiV1Test(SomeApiV2Test):
  _api_version = 1
  [..]
```

4. Change the Cinder API test directory structure

Current test directory structure is

- tempest/api/volume/ : v1 API test files
- tempest/api/volume/v2/ : v2 API test files

This structure is not understandable, and it is better to move v1 API test files to some directory which shows v1s one clearly.

This blueprint proposes to change the structure to

- tempest/api/volume/ : common test files

- tempest/api/volume/v1/ : v1 API specific test files
- tempest/api/volume/v2/ : v2 API specific test files

The test cases which can share the code between v1 and v2 will be stored in tempest/api/volume/. For specific test cases which dont have shared code, the inheritance model from (3) wont work. The hierarchy would be:

- [v1 test class] -> [CommonVolumeTest]
- [v2 test class] -> [CommonVolumeTest]

and these test files would be stored in:

- tempest/api/volume/v1/
- tempest/api/volume/v2/

Implementation

Assignee(s)

Primary assignee:
Zhi Kun Liu <liuzhikun@gmail.com>

Other Contributors:
Chandan Kumar <chkumar246@gmail.com> jun xie <junxiebj@cn.ibm.com>

Milestones

Target Milestone for completion:
Juno-2

Work Items

- Add common service client for v1 and v2
- Add a common class for Cinder v1/v2 API tests
- Add a common admin class for Cinder v1/v2 API tests
- Add new test cases using the new shared test classes Using a google docs spreadsheet to manage the task progress: (cinder_v2_api_tests)

2.1.9 Move success response checking to tempest clients

https://blueprints.launchpad.net/tempest/+spec/client-checks-success

To ensure API stability, api calls from tempest should check that the response code in the success case is the expected value. Right now these checks are done by the callers of the apis and in an inconsistent way. There is no policy or guideline for when to check. It would be better if the response code was always checked and api callers did not have to worry about this.

Problem description

It has been the policy that tests of an api should validate the response code. Failure is handled by clients raising exceptions but the caller was supposed to check the response on success. But many api tests also call other apis as part of preparing for the actual test calls. In some cases we check the response for those and in other cases we dont, but really all calls should be checked. Expecting code that calls apis to handle this is unnecessary and error prone.

The new code added to validate nova apis with schemas does the checking in the client. We should move all checking to the client except where multiple success return codes are possible.

There are two methods on RestClient that handle this:

validate_response(cls, schema, resp, body)
: This is used with the new schema validation.

expected_success(self, expected_code, read_code)
: This was an earlier attempt to rationalize response checking but is only used by the image clients.

Proposed change

Move all response checking to the clients. Use validate_response to check the success code, adding schemas for apis that do not have them yet. Then we can remove all checks from the test code itself except where there are multiple possible 2xx responses. In that case the caller should also check for a specific value.

Alternatives

Continue to clutter the test code with an inconsistent set of checks that will be required after every api call.

Implementation

Assignee(s)

Primary assignee:
: David Kranz <dkranz@redhat.com>

Milestones

Target Milestone for completion:

- Juno

Work Items

- Change all api calls in the tempest clients to check response code, adding new schemas as necessary.
- Remove all checking for 2xx from tests unless a specific value is expected out of several that could be returned by the api.

2.1.10 Refactor Client Managers

https://blueprints.launchpad.net/tempest/+spec/client-manager-refactor

The current client managers depends on a relatively large number of configuration items, that is the combination of all clients parameters. This makes its migration to tempest-lib troublesome.

Problem description

We have several plans about the client manager:

- make it as stable interface in tempest.lib. This will help in turn the migration of credential providers and service clients to the .lib namespace. And it will give people writing tempest plug-in a home for their custom service clients
- change it so that it is possible for plug-in developer to add their service clients dynamically into the client manager, so we can have a consistent way of accessing test clients from tests

The current client managers depends on CONF, and its structure does not easily allow for runtime registration of extra clients.

For instance, in *Manager* class:

```
self.network_client = NetworkClient(
    self.auth_provider,
    CONF.network.catalog_type,
    CONF.network.region or CONF.identity.region,
    endpoint_type=CONF.network.endpoint_type,
    build_interval=CONF.network.build_interval,
    build_timeout=CONF.network.build_timeout,
```

Another issue with the current structure is that new API versions lead to proliferation of client attributes in the client manager classes. With service clients being split into pieces, the size of the client manager grows accordingly.

Proposed change

Split the client manager in two parts.

The first part provides lazy loading of clients, and it does not depend on tempest CONF, as it is planned for migration to tempest.lib. It covers the six client groups for the six core services covered by tempest in the big tent. It exposes an interface to register further service clients.

Lazy loading of clients provides protection against clients that try to make API calls at __init__ time; it also helps in running tempest with the minimum amount of CONF required for the clients in use by a specific test run.

The second part passes tempest CONF values to the first one. It registers any non-core client, whether still in tempest tree or coming from a plug-in.

The client registration interface could look like:

```
def register_clients_group(self, name, service_clients, description=None,
                           group_params=None, **client_params):
    """Register a client group to the client manager

    The client manager in tempest only manages the six core client.
    Any extra client, provided via tempest plugin-in, must be registered
    via this API.

    All clients registered via this API must support all parameters
    defined in common parameters.

    Clients registered via this API must ensure uniqueness of client
    names within the client group.

    :param name: Name of the client group, e.g. 'orchestration'
    :param service_clients: A list with all service clients
    :param description: A description of the group
    :param group_params: A set of extra parameters expected by clients
                         in this group
    :param client_params: each is a set of client specific parameters,
                          where the key matches service_client.__name__
    """
```

The tempest plugin `TempestPlugin` interface is extended with a method to return the service client data specific to a plugin. Each plugin defines a new service clients group and the relevant data.

Service Clients data is stored in a singleton `ServiceClientsData`. `ServiceClientsData` is instantiated by the `TempestTestPluginManager`, which obtains the service client data from each plugin and registers it.

Client managers used by tests consume the service client data singleton, and dynamically defines a set of attributes which can be used to access the clients.

Attributes names are statically defined for now. They will be the same names as now, to minimize the impact on the codebase. For plugins, attributes names include the group name, to avoid name conflicts across service clients that belong to different plugins.

In future we may define a standard naming convention for attribute names and to enforce it by deriving

names automatically. Future names may not contain the _client suffix, to save space and allow for always specifying the client provider in test code, so to make the code more readable. This naming convention will not be implemented as part of this spec.

Alternatives

Keep the manager on tempest side, and have big tent teams write their own managers. Carry long list of clients as parameters in cred providers and other parts of tempest

Implementation

Assignee(s)

Primary assignee:
Andrea Frittoli <andrea.frittoli@hpe.com>

Milestones

Target Milestone for completion:
Newton-1

Work Items

- Core functionality in manager part1, with unit test coverage and migration of one client group
- Migration of other client groups (one per patch)
- Implementation of the registration interface (does not depend on step 2)
- Registration of non-core clients from tempest tree
- Registration of non-core clients from plugins
- Separate manager part1 into its own module, and include maanger.py along

Work has stared on this: Change-id I3aa094449ed4348dcb9e29f224c7663c1aefeb23

Dependencies

None

2.1.11 Clients Return One Value

https://blueprints.launchpad.net/tempest/+spec/clients-return-one-value

Currently tempest clients return a response code and body. Since we moved response checking to clients, almost all callers of the clients ignore the response code: *Move success response checking to tempest clients*. It would be much cleaner if clients returned a single response object that was the body, with a property to get the response status and headers if needed.

Proposed change

Introduce a new ResponseBody class in rest_client:

```
class ResponseBody(dict):
    """
    Class that wraps an http response and body into a single value.
    Callers that receive this object will normally use it as a dict but
    can extract the response if needed.
    """
    def __init__(self, response, body=None):
        body_data = body or {}
        self.update(body_data)
        self.response = response

    def __str__(self):
        body = super.__str__(self)
        return "request: %s\nBody: %s" % (self.response, body)
```

Change all the tempest clients to return this object and change all the calls to the clients, getting rid of the current _ for the response.

Alternatives

In theory this could be done in the rest client itself, rather than in each service client, but that would imply much more regularity of the service clients than we have. Also, it would then be necessary to change everything at once because all the tests currently expect two values.

The primary consideration is to not return unused values almost all the time. Another alternative would be to have an argument to the rest client methods that says whether a response should also be returned. I think the current proposal is cleaner.

Implementation

Unfortunately each client class must be changed lockstep with all calls to that client.

Assignee(s)

Primary assignees:
David Kranz <dkranz@redhat.com>

Other contributors:
Yair Fried <yfried@redhat.com>

Milestones

Target Milestone for completion:

- kilo-1

Work Items (for each service client)

- Change all methods in the tempest clients to return ResponseBody.
- Change all calls to clients to receive just the response body and check the response code if necessary.

2.1.12 Add a config file verification tool

https://blueprints.launchpad.net/tempest/+spec/config-verification

Add a new tempest tool that will query the services APIs to check if config options are set correctly.

Problem description

The tempest config file has tons of options and the number is just growing. Additionally some of the options arent exactly trivial to set completely. For example, the api extensions options are either all or a list of all the enabled extensions. Most of the services support discovery of almost all the options we use in the config file.

Proposed change

To write a tool which will use an existing config file and verify that everything is set matching what the services report as being enabled. It will start by reading in the config file and then use the tempest clients to query the services to check that things like api versions, catalog types, and extensions match what is reported by the services. It will also have a flag to overwrite the config file with the values with reported by the services.

This new tool will be added to the tools/ directory in the root of the tempest tree along with the other helper utilities in tempest.

The usage statement for the new tool will be something like the following:

```
usage: verify_tempest_config.py [-h] [-u] [-n]

optional arguments:
  -h, --help    show this help message and exit
  -u, --update  Update the config file with results from api queries. This
                assumes whatever is set in the config file is incorrect. In
                the case of endpoint checks where it could either be the
                incorrect catalog type or the service available option the
                service available option is assumed to be incorrect and is
                thus changed. A copy of the original config file will be
                created with .orig appended to the filename.
  -n, --nocopy  Don't create a copy of original config file when running
                with the update option.
```

To test the functionality of the tool unit tests will be added to verify that both the verification functionality and the config file updating work as intended.

Alternatives

The alternative would be having a tool that generated the config file directly however this has a chicken and egg problem. If you want to autconfigure tempest using feature discovery you need to have auth to talk to the services, and the auth is part of the config file. So instead of having some hybrid between a generator and existing config having a tool which verifies an existing config file would clean up any confusion. Also having an option to overwrite it with the autodiscovery results will essentially be a config generator.

Implementation

Assignee(s)

Matthew Treinish <mtreinish@kortar.org>

Milestones

Target Milestone for completion:
Juno-1

Work Items

- Add basic verification script
- Add config file updating feature
- Add unit tests for config verification functionality
- Add unit tests for config file updating functionality

2.1.13 Consistent service method names

https://blueprints.launchpad.net/tempest/+spec/consistent-service-method-names

Make service method names consistent

Problem description

Service clients are Tempest own REST clients for operating each OpenStack projects APIs. And we have a plan to migrate service clients methods to tempest-lib. However these methods names are inconsistent, and it would be difficult to use these methods from viewpoint of library users. So we need to make these names consistent and set up the way to keep them consistent before migrating.

Proposed change

Basically all methods names should be <verb>_<resource/object name>, not <resource/object name>_<verb>. There are following patterns we need to consider method names for REST API methods.

- POST /resources (Create a resource)
- PUT /resources/<id> (Update a resource)
- DELETE /resources/<id> (Delete a resource)
- GET /resources (Get a list of resources)
- GET /resources/<id> (Get the detail information of a resource)

https://etherpad.openstack.org/p/tempest-consistent-service-method-names is an investigation of current method names. Based on the investigation, this spec proposes consistent method names of each patterns. In addition, this proposes hacking checks for keeping consistent method names. The patch https://review.openstack.org/168762 is a prototype for these hacking checks. The details of them are following.

POST /resources

Naming rule: create_<resource name>

All creation methods follow this rule, so we dont need to rename creation methods. The hacking check of this rule is If a method calls self.post(), the method name should be create_<resource name>.

PUT /resources/<id>

Naming rule: update_<resource name>

All update methods follow this rule, so we dont need to rename update methods. The hacking check of this rule is If a method calls self.put(), the method name should be update_<resource name>.

DELETE /resources/<id>

Naming rule: delete_<resource name>

There are two patterns for deletion method, delete_<resource name> and remove_<resource name>. The number of delete_<resource name> is 72, and the one of the other is 11. In addition, delete_<resource name> is simple name because it is the same as HTTP method. The hacking check of this rule is If a method calls self.delete(), the method name should be delete_<resource name>.

GET /resources

Naming rule: list_<resource name>s

There are three patterns for listing resources, list_<resource name>s, get_<resource name>_list and <resource name>_list. The number of the first is 115, the one of the second is 3 and the one of the third is 2.

Some Nova APIs provide a resource list with detail information like os-hypervisors/detail and os-availability-zone/detail. These method names are get_hypervisor_list_details and get_availability_zone_list_detail now. This spec proposes these methods are merged to this naming rule like:

```
list_availability_zones(self, detail=False):
    url = 'os-availability-zone'
    if detail:
        url += '/detail'
    resp, body = self.get(url)
    ..
```

by adding the argument detail. GET /resources and GET /resources/<id> call the same method self.get() for sending a request to servers. So it is difficult to check the methods which call self.get() should be based on rules of GET /resources or GET /resources/<id>. Then this spec proposes the same hacking check for GET /resources or GET /resources/<id>. That means the hacking check is If a method calls self.get(), the method name should be show_<resource name> or list_<resource name>s.

GET /resources/<id>

Naming rule: show_<resource name>

There are two patterns for getting the detail of a resource, show_<resource name> and get_<resource name>. The number of the first is 12, and the one of the second is 126. So the number of the second is bigger. However, this spec proposes all GET /resources/<id> methods should be named to show_<resource name> because of clarifying differences from methods which are for GET /resources. There are methods for GET /resources also and some resource names are the same between a single noun and multiple nouns like chassis. So it is better to avoid using get_<resource name> for clarifying the method behavior. The hacking check of this rule is mentioned at the above.

Separate service client modules for each resource

Some service clients contain the methods for multiple resources in a single module. For example, server_client module contains the methods for /servers and /server_groups. Current separation of modules are inconsistent, and this spec proposes all service client modules will be separated into a single module by each resource. In addition, current modules of service clients contain JSON in these names but we need to remove them. Because current service clients supports JSON only and JSON in these names are meaningless now.

Implementation

Assignee(s)

Primary assignee:

- Kenichi Ohmichi <oomichi@mxs.nes.nec.co.jp>

Other contributors:

- Masayuki Igawa <igawa@mxs.nes.nec.co.jp>

Milestones

Target Milestone for completion:
Liberty

Work Items

- Rename service clients methods based on this proposal.
- Rename service clients classes based on this proposal.
- Separate service clients modules per resources.
- Add hacking rules based on this proposal.

References

- We have discussed this working items at Vancouver Summit. The log is https://etherpad.openstack.org/p/YVR-QA-Tempest-service-clients

2.1.14 Improve IPv6 API testing parity in tempest

https://blueprints.launchpad.net/tempest/+spec/ipv6-api-testing-parity

Current tempest API tests do not validate IPv6 to the same extent as IPv4.

Problem description

IPv6 is evolving in Neutron and the community is working hard to add the necessary support. However, the current API tests in tempest do not validate IPv6 to the same extent as IPv4.

Also, Neutron now supports two extended attributes for IPv6 subnets (ipv6-ra-mode and ipv6-address-mode) in the Juno timeframe.

This BP would add the necessary IPv6 tests in tempest.

Proposed change

Neutron BP: IPv6 Subnet attributes are implemented as part of the following BP - https://blueprints.launchpad.net/neutron/+spec/ipv6-two-attributes

The possible values for the subnet attributes are as follows.

- ipv6-ra-mode {dhcpv6-stateful, dhcpv6-stateless, slaac}
- ipv6-address-mode {dhcpv6-stateful, dhcpv6-stateless, slaac}

The two IPv6 attributes provide flexibility to choose the type of IPv6 network. However, not all combinations of the two attributes are valid. Valid and invalid combinations are captured in the Neutron ipv6-provider-nets-slaac.rst blueprint and also at the following link. - https://www.dropbox.com/s/9bojvv9vywsz8sd/IPv6%20Two%20Modes%20v3.0.pdf

Along with test cases related to subnet attributes, this BP would implement new test cases in tempest to bring in parity between IPv4 and IPv6 tests. To start with, new tests would be required in Neutron for Ports/Security-Groups/Subnets/FWaaS api tests.

The following etherpad link would be used to track all the test cases.

https://etherpad.openstack.org/p/ipv6-api-testing-parity

Alternatives

None

Implementation

Assignee(s)

Primary assignee:

- Sridhar Gaddam <sridhargaddam@enovance.com>
- Sean M. Collins <sean_collins2@cable.comcast.com>

Milestones

Target Milestone for completion:
Juno release

Work Items

The work items include IPv6 API test cases like

- Subnet test cases.
- Port operations including Bulk operations.
- Security Groups and Rules - https://review.openstack.org/#/c/94130
- FWaaS test cases.
- Validating if Neutron calculates and assigns IPv6 addresses properly (i.e., based on EUI-64 where applicable).

Any new test cases related to the same topic would be tracked using the following external etherpad link. https://etherpad.openstack.org/p/ipv6-api-testing-parity

Dependencies

- Neutron IPv6 Subnet attributes (ipv6-ra-mode and ipv6-address-mode) are added in the Juno+ timeframe for selecting the type of IPv6 network. Hence, a config flag needs to be added to tempest to skip the tests while running on icehouse jobs. The required changes in tempest would be addressed as part of BP ipv6-subnet-attributes.rst. https://blueprints.launchpad.net/tempest/+spec/ipv6-subnet-attributes

  Similarly, devstack needs to populate the flag during the setup. The required changes in devstack would be addressed as part of the following BP. https://blueprints.launchpad.net/devstack/+spec/tempest-ipv6-attributes-support
- Neutron: Support Router Advertisement Daemon (radvd) for IPv6 https://review.openstack.org/#/c/101306/
- Neutron: Support for IPv6 dhcpv6-stateless and dhcpv6-statefull modes in Neutron. https://review.openstack.org/#/c/102411/

2.1.15 Javelin 2

https://blueprints.launchpad.net/tempest/+spec/javelin2

During the Juno Summit we discovered that Grenade was not doing the resource validation that we believed it was. This had always been a fragile part in grenade because complex validation of resources is somewhat tough in bash. Instead we should build a tool in Tempest that provides a way to create, validate, and destroy resources for us in testing.

Problem description

We need a tool that will create a set of resources, can validate that that set of resources exists at some later point in time (temporally disconnected with no shared memory state), and can delete that set of resources. Having this tool we can very easily test that resources (users, images, servers, objects, etc) survive an upgrade undisturbed in grenade testing.

Proposed change

Create a new javelin tool in tempest as part of the cmd directory.

The usage of javelin is as follows:

```
usage: javelin.py [-h] -m <create|check|destroy>
                  [--os-username <auth-user-name>]
                  [--os-password <auth-password>]
                  [--os-tenant-name <auth-tenant-name>]
                  [--os-auth-url <auth-url>]
```

It requires admin keystone credentials to run because it must perform user/tenant creation and inspection.

Resources are specified in a resources.yaml file:

```
tenants:
  - javelin
  - discuss

users:
  - name: javelin
    pass: gungnir
    tenant: javelin
  - name: javelin2
    pass: gungnir2
    tenant: discuss

# resources that we want to create
images:
  - name: javelin_cirros
    owner: javelin
    file: cirros-0.3.2-x86_64-blank.img
    format: ami
    aki: cirros-0.3.2-x86_64-vmlinuz
    ari: cirros-0.3.2-x86_64-initrd

servers:
  - name: peltast
    owner: javelin
    flavor: m1.small
    image: javelin_cirros
  - name: hoplite
    owner: javelin
```

(continues on next page)

(continued from previous page)

```
    flavor: m1.medium
    image: javelin_cirros
```

An important piece of the resource definition is the *owner* field, which is the user (that weve created) that is the owner of that resource. All operations on that resource will happen as that regular user to ensure that admin level access does not mask issues.

The check phase will act like a unit test, using well known assert methods to verify that the correct resources exist.

This whole exercises has, and will continue to, enlighten us on ways that the Tempest rest_client is difficult to consume outside of Tempest tests. It should go a long way to making that a cleaner distinction.

Alternatives

The alternative is to fix the grenade javelin exercises, though theyve been non functional for long enough that this doesnt seem to be a fruitful direction.

Implementation

Assignee(s)

Primary::
sean@dague.net

Additional::
emilien.macchi@enovance.com

Milestones

Juno-2

Work Items

- initial pass of javelin2 that supports create / check of similar resources as were in grenade javelin
- integration of javelin2 into grenade
- addition of destroy phase to clean up javelin2
- expand # of resources beyond what was in grenade to ensure we arent failing once we get beyond singletons
- expansion of resources beyond what was in grenade - ceilometer resources - neutron resources
- unit tests in tempest

Dependencies

None

```
This work is licensed under a Creative Commons Attribution 3.0 Unported␣
↪License.

http://creativecommons.org/licenses/by/3.0/legalcode
```

2.1.16 Keystone v3 based check and gate jobs

https://blueprints.launchpad.net/tempest/+spec/keystone-v3-jobs

Check and gate jobs using keystone V3

Problem description

All check and gate jobs at the moment rely on keystone v2 as identity service. Blueprint multi-keystone-api-version-tests introduces in tempest the ability to run tests relying on keystone V3 API only. The version of the keystone API to be used is controlled via a configuration flag. Dedicated jobs are requied to exercise the V3 option, in preparation for keystone V2 deprecation planned for Juno.

Proposed change

Setup keystone v3 jobs to be run initially as experimental only. They will be promoted then to check and eventually gate.

Running fully v3 jobs is beyond the scope of tempest and infrastructure alone, as it requires changes in other OpenStack projects:

- python bindings to either support keystone v3 API or consume Keystone Client Session Objects (see http://www.jamielennox.net/blog/2014/02/24/client-session-objects/)
- core services to be integrated with keystone v3 model and API Such changes are defined in details in dedicated blueprints.

As dependencies will be implemented in Juno, it wont be possible to run full v3 jobs against Icehouse. The keystone v3 jobs can still be used against icehouse, with the limitation that only the authentication of tempest clients and creation of isolated user and projects will be based on the v3 API.

Alternatives

It would be possible to run all tests via v2 and v3 in parallel. However this would significantly increase the gate duration.

Implementation

Assignee(s)

Andrea Frittoli <andrea.frittoli@hp.com>

Milestones

Target Milestone for completion:
Juno-final

Work Items

- Define a localrc variable in devstack for auth_version=v3
- Define an option in devstack-gate to setup the localrc variable
- Define two jobs in the experimental pipeline for tempest: dsvm-keystonev3-full and dsvm-neutron-keystonev3-full
- Track the progress of dependencies, and enhance jobs accordingly
- Run the jobs on demand until all issues are fixed and results are stable
- Promote the jobs to check for tempest
- Promote the jobs to check for all projects
- Promote the jobs to gate

Dependencies

A fully v3 check job depends on having v3 support in a number of places

- tempest framework and its tests https://blueprints.launchpad.net/tempest/+spec/multi-keystone-api-version-tests
- official python bindings and CLI tools
- openstack services

The current jobs will only be partially v3 until all dependencies are met. The migration strategy from identity API v2 to v3 will be documented as part of https://blueprints.launchpad.net/keystone/+spec/document-v2-to-v3-transition

2.1.17 Tempest List Plugins Command

https://blueprints.launchpad.net/tempest/+spec/list-plugins

Provides a means to list to currently installed Tempest plugins.

Problem description

The Tempest project recently implemented a plugin system to allow external test repositories to be included in Tempest test runs in a seamless fashion. Tempest plugins are essentially Python packages that implement a specific interface and are installed via standard Python tools. However, there is not a straightforward means for knowing which plugins are currently installed.

Proposed change

Providing a means via the `tempest` command line tooling to list the installed plugins provides a consistent experience to the user. The command `tempest plugins list` would provide the user with basic information about the installed plugins:

```
> tempest plugins list
+------------+-----------------------------------------+
| Plugin     | EntryPoint                              |
+------------+-----------------------------------------+
| HelloWorld | hello_world_tempest_plugin.plugin:MyPlugin  |
| Example2   | example_tempest_plugin.plugin:ExamplePlugin |
+------------+-----------------------------------------+
```

Projects

- openstack/tempest

Implementation

Assignee(s)

Primary assignee:
slowrie dwalleck

Milestones

Target Milestone for completion:
Mitaka-2

Work Items

- Create means to query the stevedore.ExtensionManager for registered entrypoints
- Create a function that turns the list of plugins into user readable output
- Add an entry point for the `plugins list` command in the tempest.cmd package

Dependencies

- prettytable

References

- https://etherpad.openstack.org/p/mitaka-qa-tempest-run-cli
- https://github.com/openstack/tempest/blob/005ff334d485c4ca231d7ee8396d3eb979a9ce59/tempest/test_discover/plugins.py#L74
- https://github.com/openstack/tempest/tree/master/tempest/cmd

2.1.18 meta data and uuid for tests

https://blueprints.launchpad.net/tempest/+spec/meta

The purpose of this spec is to define a standard for attaching meta data such as Universally Unique Identifiers (UUIDs) to tests that are to be used by tempest. This specification should be enforced both for tests that are to live in the tempest repository as well as tests that will eventually reside within their own repositories.

Problem description

While there are many reasons to want meta data attached to tests. One of the complaints that led to this specification is the lack of a unique identifier for tests. If you cannot track a specific test over time, long term data carries less meaning and progress is more complicated to track.

Proposed change

Each test will have a decorator that will accept a required arguments containing a uuid and meta as kwargs. This open ended approach for meta data should insure flexibility for the future. Such as allow Defcore to directly tag tests with capability data moving forward.

This change will also require the development of a tool for inserting the decorators with uuid content for all the existing tests. This tool can also be used as a uniqueness validation and missing uuid checker in the gate.

Alternatives

It was also suggest that meta data can be written into parseable doc strings.

Implementation

This change will require the following components be developed.

- uniqueness tester / detect missing meta / insert meta

This tool will check for unique UUID decorators for all test_* methods in test_*py files. This is done in a single pass to build a list of undecorated tests, which can the be used as a gate check in the qa pipeline. From this list another tool will generate unique UUID metadata and insert it into all undecorated tests.

Example:

```
meta_decorator_check.py /path/to/test(s)
```

- scans the directory/files in path, parses for methods missing the decorator or finds tests with duplicate UUIDs.

With the same tool you can add the insert-missing arg. This will generate and insert the missing decorators.

Example:

```
meta_decorator_check.py --insert-missing /path/to/test(s)
```

- decorator

The UUID and meta decorator format will be as follows:

```
'@test.meta(uuid='12345678-1234-5678-1234-567812345678',
            otherdata='another value')
```

For sphinx autodoc generation it will append the following to the doc string:

```
uuid: '12345678-1234-5678-1234-567812345678'
otherdata :'another value'
```

To insure the data is passed to subunit output we add the testtools attr:

```
uuid='12345678-1234-5678-1234-567812345678'
otherdata='another value'
```

- hacking

New hacking will need to be added to insure that a decorator is set for all tests. First a new rule to be added to tempest/hacking/checks.py.

> all_tests_need_uuid_metadata
>
> Check that a test has unique UUID metadata
>
> > All tests should have a unique UUID identifier in the metadata of the form: @test.meta(uuid=12345678-1234-5678-1234-567812345678) to give a stable point of reference.

T108

The Tempest coding guide will be updated to reflect the new hacking rule.

Assignee(s)

Primary assignee:
David Lenwell (davidlenwell, dlenwell@gmail.com) Chris Hoge (hogepodge, chris@openstack.org) Sergey Slipushenko (automated test tagging)

Milestones

Target Milestone for completion:
K-3

Work Items

- creation of decorator
- creation of the generate_meta / uniqueness testing tool
- use the generate_meta tool to generate default meta data for all existing tests.
- implementation of gate tests
- update tempest coding guide with new hacking rule

Dependencies

- No known external dependencies.

2.1.19 No truncation in service clients return value and move to tempest-lib

https://blueprints.launchpad.net/tempest/+spec/method-return-value-and-move-service-clients-to-lib

Make service clients not to truncate response and move those to tempest-lib

Problem description

Service clients are Tempest own REST clients for operating each OpenStack projects APIs. And we have a plan to migrate service clients methods to tempest-lib.

1. Currently these method cut out the top key from response like:

```
def show_host_detail(self, hostname):
    """Show detail information for the host."""

    resp, body = self.get("os-hosts/%s" % str(hostname))
    body = json.loads(body)
```

(continues on next page)

(continued from previous page)

```
    self.validate_response(schema.show_host_detail, resp, body)
    return service_client.ResponseBodyList(resp, body['host'])
```

However this cutting is wrong as library function, because the caller cannot know the complete response returned from corresponding APIs.

One example is resource links which are currently truncated by service clients. So if caller needs to use those resource links, they can not get from current service clients.

2. Currently JSON schemas which are used to validate the response in service clients are present in Tempest. When service clients will be migrated to Tempest-lib, those schemas should be accessible for service clients in Tempest-lib.

Proposed change

- Stop cutting out the top key of a response we need to remove this kind of cutting from service client methods like:

```
def show_host_detail(self, hostname):
    """Show detail information for the host."""

    resp, body = self.get("os-hosts/%s" % str(hostname))
    body = json.loads(body)
    self.validate_response(schema.show_host_detail, resp, body)
-   return service_client.ResponseBodyList(resp, body['host'])
+   return service_client.ResponseBodyList(resp, body)
```

- Move JSON Response Schema to Tempest-lib Currently Tempest have JSON response schema in tempest/api_schema which are used in service clients to validate API response. During Vancouver summit, it was decided that for short term solution we can move those schema in Tempest-lib along with service clients.

  In long term, each project should provide some way to get those schema through API or something else.

- Copy the service client code to tempest-lib repository
- Switch Tempest to use the service client code of tempest-lib

Migration of service clients can be done gradually with one client class at a time.

Implementation

Assignee(s)

Primary assignee:

- Kenichi Ohmichi <oomichi@mxs.nes.nec.co.jp>

Other contributors:

- Ghanshyam Mann <ghanshyam.mann@nectechnologies.in>

Milestones

Target Milestone for completion:
Liberty

Work Items

- Modify service clients methods return value based on this proposal.
- Move JSON schema to Tempest-lib
- Move Service Clients to Tempest-lib

Dependencies

https://blueprints.launchpad.net/tempest/+spec/consistent-service-method-names

References

- We have discussed this working items at Vancouver Summit. The log is https://etherpad.openstack.org/p/YVR-QA-Tempest-service-clients

2.1.20 Check minimum version for CLI tests

https://blueprints.launchpad.net/tempest/+spec/minversion-check-for-cli-tests

There are CLI tests added to Tempest for commands which may not be available yet in released versions of the clients, so downstream packagers would not have those commands available for CI and the tests will fail.

Problem description

The use case is testing stable/icehouse server code, i.e. nova, with packaged versions of the clients that are supported for the stable/icehouse release of Nova, which is python-novaclient-2.17.0 in the 2014.1 release of the server code.

With branchless Tempest there is no stable/icehouse branch for Tempest, and so when new CLI tests are added to Tempest on master which require commands or other functions in the clients, they can fail for downstream packagers if the required commands/functions are not in released versions of the clients on pypi.python.org, where the packager may be getting their source tar.gz from.

One specific example here is the server-group-list CLI test added for the Nova client which is not in a released version of python-novaclient. Anyone running Tempest against a released version of the client will fail this new CLI test.

Note that the community gate CI does not have an issue with this since Tempest is run against trunk level code for the clients rather than released versions.

Proposed change

Add a simple decorator that can be used in the tempest/cli tests for checking that the installed version of the client is at a minimum version to support the test, otherwise the test is skipped.

The decorator would be applied to feature tests introduced since Icehouse due to the branchless Tempest strategy and the lack of a stable/icehouse branch.

Alternatives

There are not really any good alternatives to this issue for downstream packagers/deployers if they are not running CI against trunk levels of code for the clients. They can reset HEAD for Tempest to some arbitrary commit around the time of the server release they are testing, e.g. sometime around the 2014.1 release for stable/icehouse testing, but then they are frozen to that commit in Tempest and do not get future bug fixes that would have been backported when there were stable branches for Tempest. The other alternative is manually excluding the unsupported CLI tests but this is cumbersome and only works around the issue after the fact rather than putting the check in the code when the test runs.

Implementation

Assignee(s)

Matt Riedemann <mriedem@us.ibm.com>

Milestones

Target Milestone for completion:
Juno-2

Work Items

1. Write the decorator code. The work in progress patch is here:

   https://review.openstack.org/#/c/100031/

2. Apply the decorator to the CLI tests, with a primary focus on any tests added after the 2014.1 Icehouse release, especially for those tests which require newer client code than what is in a released version.

Dependencies

- This is only an issue introduced by the Branchless Tempest blueprint but is not technically tied to that blueprints implementation, but is listed here for posterity:

  https://blueprints.launchpad.net/tempest/+spec/branchless-tempest

```
This work is licensed under a Creative Commons Attribution 3.0 Unported
License.

http://creativecommons.org/licenses/by/3.0/legalcode
```

2.1.21 More Selectable Swift Tests

https://blueprints.launchpad.net/tempest/+spec/more-selectable-swift-tests

Enable to run API tests more flexibly for various Swift installation

Problem description

Currently, Tempest can select API test cases by referring discoverable_apis config setting in tempest.conf. However, this feature supports only selecting removable functions using WSGI middlewares although Swift has many functional selectabilities other than using middlewares.

Proposed change

Add config parameters in tempest.conf for selecting tests for following Swift bodys functionalities.

- (Old-style) Container Sync: mirroring objects in the container to another container
- Object Versioning: versioning all objects in the container
- Discoverability: providing details about the Swift installation

Above features are independent of middleware settings. Whether to use some middlewares or not is defined in Swifts proxy server, on the other hand, container sync and object versioning require settings in storage server and running background daemons. Discoverability function is enabled/disabled at proxy servers, but this function is to expose Swifts installed middlewares and other features, so the setting is independent of middleware settings.

Config values are added in tempest.conf as follows:

```
[object-storage-feature-enabled]
container_sync=True/False
object_versioning=True/False
discoverability=True/False
```

Implementation

Assignee(s)

Daisuke Morita <morita.daisuke@lab.ntt.co.jp>

Milestones

Target Milestone for completion:
Juno-3

Work Items

- Add config values to select tests
- Insert skip annotations into appropriate test cases

```
This work is licensed under a Creative Commons Attribution 3.0 Unported␣
↪License.

http://creativecommons.org/licenses/by/3.0/legalcode
```

2.1.22 Tempest support for multiple keystone API versions

https://blueprints.launchpad.net/tempest/+spec/multi-keystone-api-version-tests

Decouple tempest from keystone version specifics and run tests with keystone v3

Problem description

Tempest code is tightly coupled with keystone V2 specific implementations. Common classes (such as rest client and tenant isolation), test base classes and test themselves all assume the identity service is provided by a keystone v2 endpoint. Tempest shall be able to run with a keystone V3 identity service, and newer versions as they become available.

Proposed change

A new configuration flag is introduced to specify the auth version to be used. The flag is defined as follows:

```
cfg.StrOpt('auth_version',
           default='v2',
           help="Identity API version to be used for authentication "
                "for API tests."),
```

And its used to select the matching version of Credentials and Auth Provider:

```
if CONF.identity.auth_version == 'v2':
    credential_class = KeystoneV2Credentials
    auth_provider_class = KeystoneV2AuthProvider
elif CONF.identity.auth_version == 'v3':
    credential_class = KeystoneV3Credentials
    auth_provider_class = KeystoneV3AuthProvider
else:
    raise exceptions.InvalidConfiguration('Unsupported auth version')
```

A number of refactors are required to achieve this and make sure we dont need to change test code again when moving to different keystone API versions.

Authentication are factored out in an authentication provider. Credentials are handled via a dedicated class, provided to tests by a credential manager. Clients managers receive credentials and are the sole responsible for instantiating clients and provide them to tests. At the moment client managers instantiate all available clients when created. This is unnecessary, and it leads to issues when not all openstack services are available for test. Client managers are thus changed to lazy instantiation of clients.

Manager __init__ method signature before and after refactor:

```
Before:

   def __init__(self, username=None, password=None, tenant_name=None,
                interface='json', service=None):

After:

   def __init__(self, credentials=None, interface='json', service=None):
```

Authentication in rest client before and after refactor:

```
Before:

   def request(self, method, url,
               headers=None, body=None):
       if (self.token is None) or (self.base_url is None):
           self._set_auth()

       if headers is None:
           headers = {}
       headers['X-Auth-Token'] = self.token

       resp, resp_body = self._request(method, url,
                                       headers=headers, body=body)

After:

   def _request(self, method, url, headers=None, body=None):
       # Authenticate the request with the auth provider
       req_url, req_headers, req_body = self.auth_provider.auth_request(
           method, url, headers, body, self.filters)
```

Access to credentials IDs from the tests, before and after refactor:

```
Before:

       # Retrieve the ResellerAdmin tenant id
       _, users = cls.os_admin.identity_client.get_users()
       reseller_user_id = next(usr['id'] for usr in users if usr['name']
                               == cls.data.test_user)
```

(continues on next page)

(continued from previous page)

```
        # Retrieve the ResellerAdmin tenant id
        _, tenants = cls.os_admin.identity_client.list_tenants()
        reseller_tenant_id = next(tnt['id'] for tnt in tenants if tnt['name']
                                  == cls.data.test_tenant)

After:

        # Retrieve the ResellerAdmin user id
        reseller_user_id = cls.data.test_credentials.user_id

        # Retrieve the ResellerAdmin tenant id
        reseller_tenant_id = cls.data.test_credentials.tenant_id
```

Areas affected by refactor:

- Rest client (tempest/common/rest_client.py): move auth code to an external auth provider
- Client managers (tempest/manager.py, tempest/clients.py, tempest/scenario/manager.py): work with a Credentials class. Lazy load of clients.
- Tests base classes (tempest/api/**/base.py): adapt where needed to modified rest client, client manager and credentials
- Tests: adapt where needed to modified rest client, client manager and credentials

Alternatives

We could change all the code in place - without refactoring - adding checks for the configured auth version. This would still require touching a considerable chunk of tempest code, without the benefit for future keystone versions.

Implementation

Assignee(s)

Primary assignee:
Andrea Frittoli <andrea.frittoli@hp.com>

Milestones

Target Milestone for completion:
Juno-1

Work Items

- Move auth from rest_client to auth provider
- Provide unit tests for the new auth and credential classes
- Refactor Manager, Credentials class everywhere
- Client Manager provide client lazy load
- Tenant isolation support for V3
- Provide multi auth-version for API tests
- Provide multi auth-version for scenario tests
- Provide multi auth-version for CLI tests
- Provide multi auth-version for 3rd part tests
- Provide multi auth-version for stress framework
- Add experimental job with auth_version = v3

Dependencies

- Python bindings and CLI are not yet all V3 ready. Some of the work in this blueprint will have to be postponed until this is fixed

::

2.1.23 Increase the number of scenario tests for Neutron

https://blueprints.launchpad.net/tempest/+spec/neutron-advanced-scenarios

Increase the number and coverage of scenario tests in Tempest for Neutron.

Problem description

Currently there is a limited number of scenario tests in Tempest for Neutron. Attempts have been made in the recent past to increase this number. However, these efforts have not been very effective due to the following factors:

- A very small number of developers currently contributing code for Neutron scenario tests.
- The lack of a structured process to make sure developers achieve progress.

Proposed change

Under this blueprint, a structured process will be followed with the overall goal of creating a community of engaged and well supported scenario tests developers. This process consists of the following steps:

1. A How to develop scenario tests for Neutron tutorial will be developed as an extension to the Tempest documentation. This tutorial will include clear and strict guidelines for documentation and logging. On one hand, each test is unique, and complex operations will most likely be included in supporting methods or shared between modules, so documentation should provide sufficient information about the test for people outside the Tempest or Neutron efforts, hopefully even for simple users who will use the scenarios to test their deployment. On the other hand, tests operation should be logged in detail so it is clear what progress and operations took place prior to any failure that might arise.
2. A set of scenario tests will be well specified taking as a starting point the result of the design summit in Atlanta and captured at the Juno design summit etherpad. The specification of these scenarios will be done in close cooperation with the Neutron core team. The specification of each scenario will include a list of people with expertise to support the test developer.
3. A message will be sent to the openstack-dev mailing list inviting developers to select one of the scenarios specified at the Juno design summit etherpad. Developers will assign themselves to the scenarios they have interest on by signing their name next to the corresponding specification in the above mentioned etherpad.
4. **Progress will be tracked for each scenario on a weekly basis.**
    - The main goal of this tracking is to make sure developers are getting the support they need.
    - Tracking will insure developers get prompt reviews from Neutron and Tempest cores.
    - Progress will be discussed at the Neutron and Tempest weekly IRC meetings.
    - The tracking will be kept at the Juno design summit etherpad.
5. Test owners will be designated. Owners will be in charge of maintaining the code, debug future failures, enhancing code documentation and logging, as well as providing reasonable support for relevant questions (though adequate docs should minimize such questions), thus easing new contributors into the community

The tests developed as result of this process will reside in the tempest tree hierarchy at tempest/scenario

Alternatives

None

Implementation

Assignee(s)

Primary assignee:
Miguel Lavalle <miguel@mlavalle.com>

Milestones

Target Milestone for completion:
Juno-3

Work Items

- Create tutorial on Tempest scenario tests for Neutron: Juno-1
- Create specification of scenarios to develop: Juno-1
- Send message to openstack-dev inviting developers to review tutorial and select scenarios to implement: Juno-1
- Merge new Neutron scenarios to Tempest tree: Juno-3

Dependencies

None

2.1.24 Tempest Post-run Cleanup

https://blueprints.launchpad.net/tempest/+spec/post-run-cleanup

Problem description

The existing script, /tempest/stress/cleanup.py, can be used to do some basic cleanup after a Tempest run, but a more robust tool is needed to report as much information as possible about dangling objects (leaks) left behind after a tempest run in an attempt to help find out why the object(s) was left behind and report a bug against the root cause. Also the tool should completely reset the environment to the pre-run state should tempest leave behind any dangling objects.

The idea is that the user should be able look at the report generated and find the root cause as to why an object was not deleted. Also the tool will return the system back into a state where tempest can be re-run with the expectation that the same test results will be returned.

Currently there can be a good deal of manual work needed, depending on what tests fail, to return to this pre-run state. This blueprint is designed to alleviate this issue.

Proposed changes

- Keep /tempest/stress/cleanup.py as a starting point and extend it. It should be moved to from /tempest/stress to tempest/cmd/ and an entry point should be added for it as well. This way it is installed as a binary when setup.py is run, which also allows it to be unit tested. Currently the tools uses the tempest OpenStack clients and this will remain unchanged.
- Fix cleanup.py to delete objects by looping through each tenant/user, over the way it currently works, which is to use the admin user and all_tenants argument, as some object types dont support this argument, Floating IPs for example.
- Currently cleanup.py deletes all objects across all users/tenants. Add two runtime arguments: init-saved-state, that creates a JSON file containg the pre-tempest run state and preserve-state, that will preserve the deployments pre-tempest run state, including tenants and users defined in tempest.conf. This will enforce that the deployment is in the same state it was prior to running tempest and allow tempest to be run again without having to reconfigure tempest and recreate the tempest test users etc. For example, if preserve-state is true cleanup will load the JSON file (created by running cleanup with init-saved-state flag prior to tempest run) containing the preserved state of the environment and marshal the data to some defined instance variables. Then, when cleanup is looping through floating ips we would have something like:

    for f in floating_ips:

        if not preserve or (preserve and f[id] not in self.floating_ips):

            try:
                admin_manager.floating_ips_client.delete_floating_ip(f[id])

            except Exception:

- cleanup.py currently deletes servers (instances), keypairs, security groups, floating ips, users, tenants, snapshots and volumes. It should also delete any stacks, availability zones and any other objects created by Tempest, full list TBD.
- As mentioned in the overview section above some test failures leave the system in a strange state. For example, an instance cannot be deleted because it is in Error state. Even after using CLI to reset the instance to Active state, future delete calls just result in Error state once again. Such a case indicates a bug in OpenStack. This tool should should provide as much detail as possible as to what went wrong so a defect can be opened against the problem(s).
- Add argument, dry-run, that runs cleanup in reporting mode only, showing what would be deleted without doing the actual deletes

Scenario 1: run cleanup.py

This is the current behavior, which deletes all objects in the system, with the exception of the missing ones, stacks and availability zones for example.

Scenario 2: run cleanup.py preserve-state

Same as Scenario 1 except that objects defined in tempest.conf, that are used in a Tempest run are preserved.

For example (exceptions are variables defined in tempest.conf):

- delete all users except: username, alt_username, admin_username
- delete all tenants except: tenant_name, alt_tenant_name, admin_tenant_name
- delete all images except: image_ref, image_ref_alt

Additional Implications

There are cases where cruft will be left in the database do to openstack defects that dont allow objects to be removed during the cleanup process. In such cases resetting the system to the pre-existing state requires direct interaction with the database. It may be useful to design the cleanup script so that it has a pluggable interface, where downstream functionality can be added to automate required database interactions for example. Although the API delete failure indicates an upstream bug that needs to be fixed, until that bug is fixed testing the environment further is blocked until the records are deleted.

Implementation

Assignee(s)

Primary assignee:
David Paterson <davpat2112@yahoo.com>

Can optionally can list additional ids if they intend on doing substantial implementation work on this blueprint.

Milestones

Target Milestone for completion:

- Juno release cycle, approximately the week of July 24th, 2014.

Work Items

- refactor location of cleanup.py
- register new runtime arguments in cleanup.py
- enable filtering deletions based on preserve-state argument and values defined in tempest.conf
- write code for detailed reporting on dangling resources and possible root cause for cleanup failure.
- implement code for dry-run argument, report only mode.

Dependencies

Only those listed above

2.1.25 Rearrange Nova Response Schemas

https://blueprints.launchpad.net/tempest/+spec/rearrange-nova-response-schemas

Rearrange Nova Response Schemas

Problem description

Compute response schemas were implemented for v2 and v3 APIs. At that time common parts were defined in common schemas and version specific into respective directories (v2 & v3).

Now v3 API is not valid for Nova and v2 and v2.1 API's response are same. After removing v3 schemas (https://review.openstack.org/#/c/141274/) we have only 1 set of schemas for v2 (/v2.1) APIs but those end up in scattered structure.

It is difficult to read and understand API complete schema as they are defined in multiple files.

Proposed change

Rearrange current schemas into better file/directory structure and if needed then, defined schemas name more clearly as per API methods.

As Nova v2.1 APIs are current API (https://review.openstack.org/#/c/149948/), we should move all current schema files to directory name v2.1. As nova is going to release microversion also, schema files for microversion needs to go in their respective new directories.

Below are the re arrangement details-

- **Directory structure-**
  api_schema/response/compute/v2.1/ -> will contain all the schema files for v2.1.
  api_schema/response/compute/v2.2/ -> will contain all the schema files for v2.2. and so on
- **Each resource schema will be defined in single files under v2.1 directory**
  For example -hypervisors.py will have all schema of hypervisor resource API.
- Each schema name should be clear enough to easily understand the API for which they are defined. For example -
  - list_hypervisors - list hypervisors API schema
  - list_hypervisors_detail - detail list of hypervisors
  - create_get_update_<resource name> - If schema is same for create, get & update API of any resource.

  Note- Most of the schema names are defined as per above guidelines but if there are some misleading names, those needs to be fixed. For example - quota class set and quota set schemas are defined with same name (quota_set) in quotas.py and quota_classes.py.

After above re arrangement it will be easy to maintain those schemas.

Alternatives

Keep things as they are which will keep schemas readability and maintenance difficult.

Implementation

Assignee(s)

Primary assignees:
Ghanshyam Mann<ghanshyam.mann@nectechnologies.in>

Milestones

Target Milestone for completion:

- kilo-3

Work Items

- Change schema as per proposed idea.
- Import the changed schema according to their new path.
- Work will be tracked in: https://etherpad.openstack.org/p/rearrange-compute-response-schemas

2.1.26 Reintegrate Tempest-Lib

https://blueprints.launchpad.net/tempest/+spec/tempest-lib-reintegration

Problem description

For several releases weve had tempest-lib which is a separate python repo and python package that contains a stable library interface suitable for external consumption. The idea behind the project is to migrate common pieces from tempest into the new repo. However, this migration process and having code which once lived in tempest be somewhere else adds a lot of friction to the process. For example, after a service client migration if we need to update or add a test which requires a client change it requires landing a change in tempest-lib, pushing a tempest-lib release with that change included, landing a global-requirements and upper-constraints bump, and then finally we can use the change in tempest. This has proven to be very expensive process and caused a lot of headaches as weve moved more code into tempest-lib.

Proposed change

The proposed change is to copy all the library code that currently exists in the tempest-lib repo and migrate it to tempest/lib in the tempest repo. We then will declare all public interface under that directory as stable interfaces, just like we did in tempest-lib. Any public interface that lands in tempest.lib will be assumed to be a stable interface once it lives in the namespace.

The tempest-lib repository will stay around, however, instead of continuing to push migrated pieces into it or maintaining a passthrough layer with semver versioning in the tempest-lib repo well just push a final 1.0.0 release and mark the it as deprecated. Well add a python deprecation warning which will be emitted when tempest-lib is used to recommend users directly use tempest.lib. This will mean users will continue to have tempest-lib work the way it does today, but will have to migrate to its new home in the tempest repo if they want any bug fixes or features. By doing this we avoid the maintanence overhead with having to add and keep the passthrough layer. It also means were less likely to slip up by trying to implement a passthrough layer which could potentially break tempest-lib consumers. There isnt anything that would break users if they stay on tempest-lib since we will never remove the repo, we just wont have any active development there. **We will never remove the tempest-lib repo or the pypi releases for it.** We just likely will never push another release or any patches to it post deprecation and re-integration.

This reintegration of tempest-lib should change our release cadence for tempest a bit. Previously weve pushed tempest releases on every start of a new OpenStack release, a major new feature change lands in tempest, and when we EOL a stable branch. However, since well likely be more frequently adding things to the library interface bumping the version will likely happen more frequently. The release versioning scheme for tempest will change slightly. Well still keep the monotonically increasing integer, but well also have a minor versions to indicate tempest-lib versions in a semver-like manner inside that. For example, tempest-10.0.0 will indicate the first release in the series with mitaka support. Then 10.1.0 will be the first tempest.lib release in that series with feature adds, and 10.0.1 would be a tempest.lib bugfix release after the mitaka release (but before the feature release). The minor versions will be reset to 0 at the start of every major version. Continuing from the previous example, at the Kilo EOL the version will be 11.0 regardless of how many tempest.lib version bumps we pushed.

Tempest-lib migrations which are really efforts to stabilize the interfaces are still valuable. But, instead of going through the process of migrating all the code to an external repo we just have to move the code to tempest.lib in the tempest repo. Then we can add the tempest-lib shim if desired.

Alternatives

Maintain the status quo

We can keep the status quo. This works for the most part, but it adds a lot of friction to the development workflow. Especially as we try to mature or add on to existing interfaces. The perfect example of this is with anything involving the service clients. The OpenStack APIs evolve over time (hopefully using microversions) and we need to update the clients to use the new features in tests this becomes a lengthy and drawn out process.

Create a passthrough tempest-lib wrapper

We could keep tempest-lib around, but instead of holding code directly it could be just be a passthrough to tempest.lib. This would be done for 2 reasons, backwards compatibility for current tempest-lib users and to enable using semver versioning of the library interface.Tempest-lib could become a library with a shim passthrough to tempest.lib for all the modules exposed via the library interface. Wed then start the 1.x.x release series for tempest-lib after the lib code has been moved back to tempest. Tempest-lib would depend on tempest in requirements.txt and we could control the tempest versions used by the passthrough in tempest-libs requirements. For example, the rest_client module in tempest-lib would end up looking something like:

```
from tempest.lib.common import rest_client

__all__ = ['RestClient', 'ResponseBody', 'ResponseBodyData',
           'ResponseBodyList']

class RestClient(rest_client.RestClient):
    pass

class ResponseBody(rest_client.ResponseBody):
    pass

class ResponseBodyData(rest_client.ResponseBodyData):
    pass

class ResponseBodyList(rest_client.ResponseBodyList):
    pass
```

Implementation

Assignee(s)

Primary assignee:
Matthew Treinish <mtreinish@kortar.org>

Milestones

Target Milestone for completion:
Mitaka Release

Work Items

- Take ownership of tempest on pypi[1]
- Enable publishing tempest to pypi[2]
- Add reno release notes to both tempest and tempest-lib
- Iterate through reintegration: * Move all the code from tempest-lib to tempest.lib in the tempest repo
- Add tempest docs about the lib interface and the new release versioning
- Push new tempest release to mark reintegration of lib
- Add python warning for tempest-lib deprecation
- Push tempest-lib release 1.0
- Modify the existing migration tooling to work with the new lib location

Dependencies

This shouldnt be dependent on any other effors, however it may cause conflicts with other BPs in progress, especially with in-progress efforts to do lib migrations.

References

2.1.27 Resource Cleanup

https://blueprints.launchpad.net/tempest/+spec/resource-cleanup

Tempest test resource cleanup

[1] http://sourceforge.net/p/pypi/support-requests/590/
[2] https://review.openstack.org/#/c/275958/

Problem description

The cleanup/release of test resources created/allocated in the class level test fixtures is invoked in the class level tearDown fixture. However tearDownClass is invoked by the unittest framework only in case setUpClass is successful. This is causing resources being leaked when:

- a skip exception is raised after resources (typically test accounts) have already been allocated
- there is a temporary failure in the system under test which causes the setUpClass to fail

The test-accounts bp introduces the possibility to run parallel tests using a configured list of pre-provisioned test accounts. Test accounts are allocated and released by each test class, and a failure to release leads to exhaustion of test accounts.

Proposed change

Disallow overriding `setUpClass` defined in `BaseTestCase` with a hacking rule. Define `setUpClass` so that it calls one (or more) other methods to be overridden by descendants. Decorate it with the `@safe_setup` decorator. This way tearDownClass will always be invoked.

```
@classmethod
@safe_setup
def setUpClass(cls):
    cls.setUpClassCalled = True
    cls.resource_setup()
```

While doing this change, an extra benefit can be gained by structuring the setup in a series of methods, to enforce the least possible resource allocation before failure and thus quick cleanup as well.

PoC for this is available here: https://review.openstack.org/#/c/115353.

```
@classmethod
@safe_setup
def setUpClass(cls):
    cls.setUpClassCalled = True
    # All checks that may generate a skip
    cls.setup_skip_checks()
    # Any setup code that does not require / generate test resources
    cls.setup_pre_resources()
    # Allocation of all required credentials
    cls.setup_allocate_credentials()
    # Shortcuts to clients
    cls.setup_clients()
    # Allocation of shared test resources
    cls.setup_create_resources()
    # Any setup code to be run after resource allocation
    cls.setup_post_resources()
```

The tearDownClass fixture requires fixing in several places, because several tearDownClass implementation would become unsafe, as they expect attributes defined during setUpClass, which may not be there anymore.

Disallow overriding `tearDownClass` defined in `BaseTestCase` with an hacking rule. Define `tearDownClass` so that it invokes a descendant specific cleanup code, and finally cleans-up credentials.

```
@classmethod
def tearDownClass(cls):
    at_exit_set.discard(cls)
    try:
        cls.resource_cleanup()
    finally:
        cls._cleanup_credentials() # Defined in BaseTestCase
```

Alternatives

Two alternatives have been identified.

Massive fixture decoration

Decorate all `setUpClass` implementation with `@safe_setup` and all `tearDownClass` implementation with `@safe_teardown`. This approach requires a mass change to tempest, which as the benefit of being almost scriptable (PoC: https://review.openstack.org/#/c/115123/). It has the downfall of requiring every new test class to add those two decorators.

Migrate to TestResources

This may still be an option on the long term, but at the moment the effort of the migration would be more than the benefit from it. Additional work to ensure cleanup of resources would still be required anyways.

Implementation

Assignee(s)

Andrea Frittoli <andrea.frittoli@hp.com>

Milestones

Target Milestone for completion:
Juno-final

Work Items

- Define base fixtures
- Migrate base classes and tests to use the new framework (multiple patches) Work tracked in https://etherpad.openstack.org/p/tempest-resource-cleanup
- Hacking rule to prevent overriding of setUpClass and tearDownClass

Dependencies

None

2.1.28 Multiple strategies for ssh access to VMs

https://blueprints.launchpad.net/tempest/+spec/ssh-auth-strategy

Different strategies for ssh access to VMs in tests.

Problem description

Ssh access to created servers is in several cases key to properly validate the result of an API call or a scenario (use case) test. This is true for compute but not limited to it. Network and volume verification must often rely on test servers, and ssh access to the VM helps significantly for the verification.

Support for ssh access to VMs in tempest tests is both heterogeneous as well as incomplete. Not all tests honour the same config options. The existing `run_ssh` option is only taken into account by some of the tests, the compute API ones. Not all tests use the same strategy for ssh access, and several tests do not perform any ssh verification at all. The reason often is that ssh verification is a common source of flakiness and timeouts in tests, and allocation of the resources required for ssh verification can be expensive.

Proposed change

Consolidate the available configuration options and make sure they are honoured everywhere. Configuration shall be declaritive, i.e. tempest users shall configure how they expect ssh to work, and if thats not compatible with the deployed cloud tempest shall raise an `InvalidConfiguration`. Improve the configuration help text to guide configuration for instance validation.

Current configuration options relevant to instance validation are:

- `CONF.auth.allow_tenant_isolation`: affects the fixed network name
- `CONF.compute.[image|image_alt]_ssh_user`
- `CONF.compute.image_ssh_password`: not image specific, and its used by only two tests, without checking against the ssh_auth_method
- `CONF.compute.image_alt_ssh_password`: unused
- `CONF.compute.run_ssh`
- `CONF.compute.ssh_auth_method`: used for resource setup by API compute tests, but not honoured by the tests. The image[_alt]_ssh_[user|password] settings are meant to be used when this is set to configured. At the moment it is not enforced nor documented
- `CONF.compute.ssh_connect_method`: used for resource setup by API compute tests, not honoured by the tests. When set to floating, it should be verified that a floating IP range is configured

- CONF.compute.ssh_user: currently used for ssh verification by most API and scenario tests, which is a problem because configuration supports different images, each with an own ssh user
- CONF.compute.ping_timeout: used by scenario test only
- CONF.compute.ssh_timeout: used by RemoteClient
- CONF.compute.ssh_channel_timeout: used by RemoteClient
- CONF.compute.fixed_network_name: used by API and scenario tests. Its the name of the network for the primary IP with nova networking; or with neutron networking when tenant isolation is disabled. The logic, as implemented by test_list_server_filters shall be moved to an helper and reused everywhere. It may be used for ssh validation only if floating IPs are disabled
- CONF.compute.network_for_ssh: used by RemoteClient and some scenario tests to discover an IP for ssh validation. It can be used if floating IP for ssh is disabled, in which case the fixed_network_name could be used as well; except for the case of multi-nic testing, which would require more logic anyways to enable the 2nd nic
- CONF.compute.ip_version_for_ssh: used by RemoteClient. It should be overridable via parameter instead of one config for all tests.
- CONF.compute.use_floatingip_for_ssh: used by some scenario tests, duplicate of ssh_connect_method, which is not used at the moment
- CONF.compute.path_to_private_key: unused
- CONF.network.tenant_network_reachable: used by scenario tests. In some cases its used for tests that want to verify both tenant and public network connectivity. In other cases its used to find out which IP to be used for instance validation, which overlaps with the ssh_connect_method
- CONF.network.public_network_id: used for allocation of floating IPs when neutron is enabled.

Target configuration shall include a new group validation used for all option related to validation of API call results, and the following options:

- CONF.validation.connect_method: default ssh method. Tests may still use different method if they want to do so (fixed or floating)
- CONF.validation.auth_method: default auth method. Tests may still use a different method if they want to do so (only ssh key supported for now). Additional methods will be handled in a separate spec
- CONF.validation.ip_version_for_ssh: default IP version for ssh
- CONF.validation.*timeout (for ping, connect and ssh)
- CONF.*.*ssh_user (for the various images available)
- CONF.network.fixed_network_name: default fixed network name; this parameter is only valid in case of nova network (with flat networking), and for now with pre-provisioned accounts. Once the bp test-accounts-continued is implemented this may still be used as default fixed network name if not specified in accounts.yaml.
- CONF.network.floating_network_name: default floating network name, used to allocate floating IPs when neutron is enabled. Deprecates CONF.network.public_network_id
- CONF.network.tenant_network_reachable: used when the configured ssh_connect_method is fixed. If this is set to false raise an InvalidConfiguration exception

Configuration options that are renamed or that planned for removal should go through the deprecation process.

A few options are image specific: image name, ssh user / password, typical time to boot / ssh. Such options would be better handled in a dedicated images.yaml file rather than in tempest.conf. This will be handled in a separate spec.

Define an helper functions that read, validate and process the configuration, which in future will help decoupling `create_test_server` from CONF, for migration to tempest-lib.

Extend the existing `RemoteClient` to provide tools for:

- ping: attempts a single ping to a target to server
- connect: attempts a single TCP connect on a generic port to a target server
- ssh: attempts a single ssh connection to a target server
- validaton: validates a server by using a configurable sequence of the above; cares about retries and timeouts

Bits of implementation for that are already available in scenario tests. They should be consolidated in `RemoteClient`.

Define a `validation_resources` function, similar to the existing `network_resources`, to be used in the class level `resource_setup`, which allocates required reusable resources, such as: a key pair, a security group with rules in it, and a floating ip. It returns all the resources in form of a dict, ready to be used in `create_test_server`. Tests which use more than one server will allocated additional floating IPs on demand. Once bp test-accounts-continued is implemented as well we may consider consolidating `validation_resources` and `network_resources`.

Centralize `create_test_server`, and make sure all tests use this central implementation. Add the following features:

- it includes an `sshable` boolean parameter in the `create_test_server` helper function, defaults to `False`. If set to `True` it ensures the server is created with all the required resources associated, e.g. that it has a public key injected, and IP address on a public network, a security group that allows for ICMP and ssh communication. The default to false ensures that resources are used only when required.
- it accepts a resources dict with reusable items, which can be: a key_name, a security_group with rules for ssh and icmp in, a floating_ip. These are passed in as parameters in preparation for the migration to tempest-lib.
- it extends the valid value for `wait_until` with new types of wait abilities: `PINGABLE` and `SSHABLE`. For instance if an `SSHABLE` server is requested the create method takes care of performing basic ssh validation as well.
- it returns a tuple `(created_server, remote_client)`, where the remote client is already initialized with access resources such as public key, admin password, IP address, ssh account name.

```
def create_test_server(self, client, wait_until=None, sshable=False,
                       resources=None, **kwargs):
    if sshable == True and run_ssh == True:
        read config via helpers
        process result, extend kwargs, but do not override
            public_key: if key_name not defined use from resources or create
            sg rules: use from resources, or create sg with rules and append
```

(continues on next page)

(continued from previous page)

```
            network name: append to network dict
            floating ip: use from resources or allocate one
        validation == True
    (...)
    server = servers_client.create_server(**kwargs)
    wait for status
    if ip_type == 'floating':
        attach an IP
    if validation:
         build params based on helpers above
         remote = RemoteClient(**params)
         wait for status (extended: ping / connect / ssh)
         return remote

def test_foo(self):
    myvm = servers.create_test_server(
        sshable=True, wait_until='SSHABLE')
    myvm['remote_client'].write_to_console("I could do something more useful")
```

A server can still be made ssh-able by-hand for more complex scenarios, such as hot-plug tests, where the server may only be connected at a later stage to a public network.

In case a test class contains tests which make use of ssh-able servers, network resources must be prepared for the tenant (if not yet available), so that it is possible to have network access to the VM.

Alternatives

As run_ssh is currently disabled, an alternative could be to completely drop ssh verification from API tests. However a number of cases cannot really be verified unless ssh verification is on (e.g. reboot, rebuild, config drive).

Implementation

Assignee(s)

Primary assignee:
Andrea Frittoli <andrea.frittoli@hp.com>

Other assignees:
Nithya Ganesan <nithya.ganesan@hp.com>, Joseph Lanoux <joseph.lanoux@hp.com>

Milestones

Target Milestone for completion:
Kilo-2

Work Items

- Introduce new configuration options, and helpers to read them
- Create a validation_resources function
- Create shared create_test_server function
- Create shared ssh verification function / extend RemoteClient
- Migrate tests to the new format (multiple patches)
- Deprecate un-used / removed configuration options
- Setup experimental / periodic jobs that run with validation enabled - the aim is to promote both run_ssh and sshable to be `True` by default, as well maintain the code path healthy until that happens

Dependencies

None

2.1.29 Tempest CLI Improvements

https://blueprints.launchpad.net/tempest/+spec/tempest-cli-improvements

Make the Tempest CLI design more consistent and intuitive by utilizing the setuptools and cliff python libraries.

Problem Description

There are currently some Tempest CLI endpoints created when tempest is installed but there is no consistency in the console command names or function.

Proposed Change

Create an intuitive set of CLI endpoints for Tempest.

Add cliff Support to Tempest

Cliff enables creation of console scripts by using a clean class structure for building applications and commands.

See: https://pypi.org/project/cliff/

For example setup.cfg would have:

```
[entry_points]
console_scripts =
    tempest = tempest.cmd.main:main
tempest.cm =
    cleanup = tempest.cmd.main:CleanupCmd
    # ...
```

and tempest.cmd.main would look something like:

```
from cliff.app import App
from cliff.commandmanager import CommandManager
from cliff.command import Command
# ...

class Main(App):

    log = logging.getLogger(__name__)

    def __init__(self):
        super(Main, self).__init__(
            description='Tempest cli application',
            version='0.1',
            command_manager=CommandManager('tempest.cm'))

    def initialize_app(self, argv):
        self.log.info('tempest initialize_app')

    def prepare_to_run_command(self, cmd):
        self.log.info('prepare_to_run_command %s', cmd.__class__.__name__)

    def clean_up(self, cmd, result, err):
        self.log.info('Tempest app clean_up %s', cmd.__class__.__name__)
        if err:
            self.log.info('Tempest error: %s', err)

# A sample command implementation would look like:

class CleanupCmd(Command):
    log = logging.getLogger(__name__)

    def get_description(self):
        description = "Utility for cleaning up ..."
        return description
```

(continues on next page)

(continued from previous page)

```
    def get_parser(self, prog_name):
        parser = super(CleanupCmd, self).get_parser(prog_name)
        parser.add_argument('--init-saved-state', action="store_true",
                            dest='init_saved_state', default=False,
                            help="Init help...")
        # More args ...
        return parser

    def take_action(self, parsed_args):
        cu = cleanup.Cleanup(parsed_args)
        cu.run()
        self.log.info("Cleanup Done!")

The end result, after running 'setup.py install', this command is valid::

    tempest cleanup --init-saved-state
```

Proposed command structure

```
tempest cleanup
    arguments:
        --dry-run
        --init-saved-state
        --delete-tempest-conf-objects

tempest create-config
    arguments:
        TBD

tempest verify-config
    arguments:
        --update, -u
        --output, -o
        --replace-ext, -r

tempest javelin
        --mode, -m
        --resources, -r
        --devstack-base, -d
        --config-file, -c
        --os-username
        --os-password
        --os-tenant-name
```

Implementation

Assignee(s)

Primary assignees:
David Paterson

Milestones

Target Milestone for completion:
Liberty-1

Work Items

- Add support for Cliff.
- Define endpoints and commands in setup.cfg.
- Create stubbed tempest.cmd.main module providing main cliff-based CLI facade.
- Refactor and migrate existing commands. For each command a new class that extends cliff.command.Command will need to be implemented:
  - javelin2
  - run-tempest-stress
  - tempest-cleanup
  - verify-tempest-config
- Migrate config_tempest.py from downstream repository and integrate with cliff.

Dependencies

- cliff - adds framework for creating CLI applications and commands.

References

- https://etherpad.openstack.org/p/tempest-cli
- https://etherpad.openstack.org/p/YVR-QA-Tempest-CLI
- https://etherpad.openstack.org/p/YVR-QA-Liberty-Priorities
- https://docs.openstack.org/cliff/latest/
- https://github.com/redhat-openstack/tempest/blob/master/tools/config_tempest.py

2.1.30 Tempest Client for Scenario Tests

https://blueprints.launchpad.net/tempest/+spec/tempest-client-scenarios

Tempest currently has tests using 2 different OpenStack clients. The first is a client written in Tempest for testability and debugability. The second is the various native clients. This adds debt to the Tempest code that we should remove.

Problem description

As Tempest grew up we grew tests that included poking directly at the raw API with our own client, as well as through the various native clients for the projects. As the volume of tests have grown, and some of the complexities in Tempest (like tenant isolation) have shown up, the 2 client strategy has become problematic.

1. It means that various abstractions need to be built above the clients to do things like waiting for resources to be created, handling tenant isolation, and doing safe cleanup.
2. The debugging output is radically different depending on the client that has failed. We can fix and react to a debuability issue in the Tempest client in tree. Addressing something as simple as reduction of extraneous token messages needs to be landed in 10 trees before its fixed in a tempest run.
3. Its demotivating to work on the code.

Proposed change

We do a wholesale cut over of the openstack clients to the Tempest client in all the scenario tests.

We remove the abstractions that were built just for these clients.

Alternatives

Keep things as they are. This however has begun to be a top issue impacting gate debugability.

Implementation

Assignee(s)

Primary assignee:

- Masayuki Igawa <igawa@mxs.nes.nec.co.jp>

Other contributors:

- Andrea Frittoli <andrea.frittoli@hp.com>
- Daisuke Morita <morita.daisuke@lab.ntt.co.jp>

Work Items

- replace official clients in tempest/scenario with tempest clients
- add hacking rule to provide use of official clients
- remove tenant isolation abstraction

Will be tracked in:
https://etherpad.openstack.org/p/tempest-client-scenarios

Dependencies

None

Referenences

Mailing list discussion - http://lists.openstack.org/pipermail/openstack-dev/2014-July/039879.html

2.1.31 Tempest External Plugin Interface

https://blueprints.launchpad.net/tempest/+spec/external-plugin-interface

Create an external plugin interface to enable loading additional tempest-like tests from out of tree to enable projects to maintain their own testing out of tree.

Problem description

As part of the recent governance change to rework OpenStack under a big tent the QA program needs to be able to handle an influx of OpenStack projects. As part of this pivot most QA projects are shifting from directly supporting every project in-tree to moving towards providing projects to self service. A missing piece of this was the tempest external plugin model, all the other QA projects contain support for extending functionality with out of tree plugins but tempest was putting the burden on the user for constructing this. Moving forward having a unified supported model for enabling this is necessary.

Proposed change

The first step for the plugin interface is to handle the loading of the additional tests in the plugin. The additional paths of the external tests will need to be passed into the unittest discovery mechanism. To do this registering an entry point that tempest will be able to load as part of its run command will be used. This will enable any installed python package which contains a tempest entry point to be seamlessly discovered and used as part of a wider Tempest test suite.

You would register the entrypoint when using pbr by adding an entry to your setup.cfg like:

```
[entry_points]
tempest.test_plugins =
    plugin_name = plugin_dir.config:load_plugin
```

This will register a new plugin to tempest to use the additional testing. Stevedore will probably be leveraged on the tempest side in the new cli run command to load the plugins when run to add the additional pieces. The hook referred to in the setup.cfg will return the test path to use for unittest discovery. The stevedore manager in tempest will then use this returned path to construct a set of test paths to use for discovery including tempest in-tree tests and all installed plugins, this will give the appearance when running tempest of a unified test suite.

As for the plugins themselves they will contain a few key pieces, mainly the tests, additional configuration.

An example of the base directory structure for a plugin to start with would look something like:

```
plugin_dir/
  config.py
  tests/
    api/
    scenario/
  services/
```

However, this is just suggested bare minimum, it is not going to cover every potential plugins use case so have additional files or directories in plugins will be expected.

The various pieces here:

- config.py will contain 2 things first is the registration method which will provide all the necessary information to tempest to execute the plugin remotely and second the additional configuration options. There will be a unified method on the plugin class, likely called register_opts(), (and also list opts so that we have a call for sample config generation) which will register all the new groups and new options. This puts the burden on the plugin to either create new groups for options or to extend existing groups where appropriate.
- The tests dir will contain the actual tests. The example above used 2 subdirs for api tests and scenario tests, but that isnt a hard requirement. Any desired organization of tests can be used in a plugin. However a single parent dir for all tests is used to make the test discovery logic in tempest a bit easier to deal with.
- Services will contain the additional api clients based on the RestClient from tempest-lib if needed.

It is also worth noting that we should strive to make these plugins be able to fully self execute on their own with a traditional test runner without any need to use tempest. The use of the plugin is only to enable easier integration with the wider test suite for other projects. A cookiecutter repo (either as part of the existing devstack-plugin-cookiecutter or a separate repo) will be leveraged to enable a fast path template for projects to use when initially setting up a new tempest plugin.

There are several things from tempest-lib which are missing which are really needed to leverage out-of-tree plugins, mainly the base test class fixtures and the credential providers. These are not requirements for starting work on the plugins as the plugins can temporarily depend on tempest code for that however, in the medium term we should migrate these to tempest-lib to lock down the interface for using them.

Another thing which well be changing is our contract around the tempest config file. Traditionally we have said not to rely on config options from directly from tempest but this will change with plugins since code out of tree will have to rely on existing options as a base set to build off of. This means in the future well have to provide better guarantees on the stability of tempests config file.

Alternatives

An alternative approach would be to not use loadable entrypoint and instead place the burden on the tempest end user to load the plugins. So when calling tempest run you would specify a list of plugins to load at the same time. The disadvantage of this approach is that it requires the user to know where the plugins are located on the system. While using an entrypoint only requires the additional code to be installed and then tempest will use external location automatically.

Additionally, if a plugin interface was not added there would be nothing stopping anyone from using tempest-lib and the same mechanism as what is expected in a plugin to create an isolated tempest-like test suite. (in fact several projects have already done this) This was also the previously recommended approach for out of tree tempest tests. The disadvantage with only doing this is that everything is treated in isolation so each test suite would have to be dealt with in isolation. However, with the scope of whats allowed in tempest clearly defined now we expect many more of these external suites to be added in the future. Being able to deal with them in a single manner and using a single workflow to deal with all of them at once is much more desirable, and a far lower burden on the end user.

Projects

- openstack/tempest
- openstack/tempest-lib

Implementation

Assignee(s)

Primary assignee:
mtreinish

Milestones

Target Milestone for completion:
Liberty-2

Work Items

- Add support to tempest run to load installed entry-points
- Modify the tempest documentation to outline project scope and changes to in-tree tempest policy (including things like config file changes)
- Create documentation for using plugins
- Add missing interfaces to tempest-lib
- Create cookie-cutter repo for tempest-plugins

Dependencies

- A tempest unified cli which adds a new run interface is required before we have a place to add the extension loading support to
- Adding an additional interface to unittest to handle a list of discovery points might be needed.

2.1.32 Create Separate Functional Testing Library

https://blueprints.launchpad.net/tempest/+spec/tempest-library

With a recent desire to create a number of functional tests for individual OpenStack projects a common toolkit for building functional tests is needed. Using Tempests core functionality start a new functional testing library to reuse this code in spinning up new project specific test suites.

Problem description

For several years Tempest has been the integrated test suite for OpenStack. However, recently as the scope and complexity of Tempest have continued to grow weve found that there is a need for project specific functional testing. With the ease of spinning up devstack slaves for testing this is a simple thing to add to the ci infrastructure. However writing the functional test suite is not as straightforward and would require a great deal of duplicated effort between the projects.

Proposed change

Take what common test infrastructure we have currently in tempest and split it out into a separate library. This library will live in a separate repository, published on pypi, have a separate bug tracker, and enforce a stable api for use. (which will be enforced with unit testing) The intent is for this library to be usable to build a functional test suite using the same building blocks as Tempest.

As functionality is moved from Tempest into the library it may need to be refactored slightly to be portable. For example, the RestClient and the service clients will need to be refactored to not depend on the tempest config file. These changes should occur at the same time the functionality is added to the library.

Once a set of functionality is ported to the library it can be removed from tempest tree and the library used instead. We should do this as soon as the functionality lands in the library. This will ensure that we arent maintaining 2 sets of the same code. It will also help ensure the functionality of the code by actually using the library interface inside of tempest. For example consider this workflow with python-novaclient CLI tests:

1. Add CLI base test classes with core functionality to the new library
2. Switch the tempest CLI tests to use the library instead
3. Remove code in tempest which has been switched to the library
4. Add a functional test suite to the novaclient repository and copy the appropriate CLI tests from tempest
5. Remove the copied novaclient CLI tests from tempest

The first 3 steps from this example will be applied to anything that moves into the library. Steps 4 and 5 only apply when weve decided a certain class of testing no longer belongs in tempest.

The first working example of this should be the CLI tests because they were only added to Tempest before having non-Tempest devstack jobs was a simple matter. By there very nature they should be a project specific functional test so adding CLI framework to the library should make it quite simple to move those tests out of Tempest.

The other aspect to consider is that the library will be consumed by the projects functional tests, not mandating functionality. All the pieces should be optional, so that projects can use what they need to. For example, while the tempest clients will be available in the library there should be no requirement to use it. Also, until decisions are made around removing a certain class of testing from tempest we shouldnt be removing tests from tempest.

Alternatives

One alternative is that we modify code inside of Tempest to do the same basic functionality. The issue with this is that tempest will become overloaded with having too many jobs. Additionally, the new library could conceivably contain features and code that would have no place inside of Tempest, but would belong in the project specific functional testing. Additionally, splitting it out into separate library will make the publishing and release a simpler matter, since we wouldnt necessarily want to be publishing all of tempest on pypi as a test-requirement for the projects.

Implementation

Assignee(s)

Primary assignee:
Matthew Treinish <mtreinish@kortar.org>

Other contributors:
Kenichi Ohmichi <oomichi@mxs.nes.nec.co.jp>

Milestones

Target Milestone for completion:
Kilo Release

Work Items

- Start new repository for the library
  - Create launchpad page for the project
  - Create PyPI entry for the project
- Copy base test class and other core functionality to run tests
- Copy and remove config and tempest specific code from CLI test framework
- Add devstack support for installing the library from git
- Setup the infra jobs to make it co-gating along with all the projects that consume it
- Migrate other commmon features and utilities related to testing, for example:

  - Exceptions
  - Common decorators
  - Matchers
  - SSH validation code
- Copy and convert the Tempest REST Client
  - Clean up the REST Client code
  - Separate the base REST Client code from Tempest specific code
  - Move the REST Client specific exception to base REST Client code
  - Copy the base REST Client code to tempest-lib repository
  - Switch Tempest to use the base REST Client code of tempest-lib
- Cleanup in Tempest service clients
  - Use ResponseBody/List on all service clients for consistent interface
  - Remove CONF values from service clients

    Current service clients contain CONF values from tempest.conf but they should be independent from tempest.conf as library functions.
- Add documentation and examples for using the libraries interfaces

Dependencies

This shouldnt be dependent on any other effors, however it may cause conflicts with other BPs in progress, so care should be made when porting things to ensure all the in progress efforts dont end up being lost in the aftermath of a library conversion.

References

- http://lists.openstack.org/pipermail/openstack-dev/2014-March/028920.html

2.1.33 Tempest Run Command

https://blueprints.launchpad.net/tempest/+spec/tempest-run-cmd

Describes a domain-specific `tempest run` command to be used as the primary entry point for running Tempest tests.

Problem Description

There are a wide range of Tempest use cases ranging from OpenStack gate testing to the testing of existing public and private clouds across multiple environments and configurations. Each of these user scenarios has its own requirements and challenges.

Currently, tempest doesnt actually control its own execution. It instead relies on an external test runner, it does not provide a consistent experience for consumers of Tempest. Users also often have impression that tempest controls its own execution. In addition, because these test runners are in no way specific to Tempest any, items that are domain specific (such as configuration) must be performed out-of-band using shell scripts or other means.

Proposed Change

Since an effort is already underway to create a set of Tempest-specific command line tooling, this spec further defines a `tempest run` command. This spec addresses the following problems:

- Providing a flexible runner that enables multiple approaches to the test discovery and execution processes
- Facilitating ease of configuration and execution of Tempest across multiple environments and configurations
- Builds on testrepository directly in order to leverage current and future testrepository capabilities

This spec outlines the beginning steps and basic functionality of the initial implementation of the run command. It is expected that the functionality of run will grow over time to suit the needs of tempest in the future.

The command implementation can be broken down into three components:

- Converting the selection regex logic from ostestr into a reusable module
- A command line interface that users will interact with
- A client that drives the execution of tests by interfacing directly with testrepository

The logical flow of the proposed test runner is as follows:

- Parse any command line arguments.
- Set necessary environment variables for Tempest based on inputs.
- Determine the set of tests to run using ostestr regex builder
- Call into testrepository with the testr-specific arguments
- Receive the results from test execution
- Perform any post-processing on the test results, if applicable.

Command Line Interface

The list of proposed command line arguments are as follows:

Test Execution:

```
--parallel
--serial
--workers <workers>
--list
```

Test Selection and Discovery:

```
--tags <list of tag name>
--services <list of services>

--include <regex>
--whitelist-file <file name>
--exclude <regex>
--blacklist-file <file name>
  Sample regex file:

      (^tempest\.api) # Comments about this regex
      tempest.scenario.test_server_basic_ops # Matches this test explicitly
```

Aliases for most commonly used regexes:

```
--smoke
--all
```

By default the regex will run the equivalent of the full jobs in tox. (running everything but tests tagged as slow)

Output:

```
--subunit
--html <file name>
```

By default the console output will be output from subunit-trace

Tempest Configuration:

```
--config <config file>
```

Part of having tempest run having domain specific knowledge is that its aware of tempest workspaces and when running in it. However, workspaces arent a requirement for actually running tempest, and there is are existing workflows where you have a separate tempest config file. (which previously could only be specified by environment variables) This option is providing an easier to use place on the CLI for doing this. This is a key advantage of having tempest own its runner is that it provides another place for passing this type of information into tempest which we previously could only do via env vars or the config file.

Testrepository Integration

One of the goals of this spec is to develop an entry point from Tempest that integrates directly with testrepository rather than calling out to testr via subprocess. This integration is a more robust design that allows new features in testrepository to propagate more easily to the Tempest runner. Inversely, as the Tempest runner evolves, features that would be useful to any test runner can be pushed down the stack into testrepository.

The planned integration point of the tempest run command with testrepository is the CLI UI for testr. However, this only one possible approach. The final solution is likely to evolve during development.

Projects

- openstack/tempest

Implementation

- Extract the regex building logic from ostestr into an externally consumable module
- Create a `tempest run` entry point in Tempest using cliff
- Handle setup of Tempest specific options such as Tempest configuration
- Implement test selection logic using the ostestr bits and based on the provided filtering options (regexes, tags, etc.)

Assignee(s)

Primary assignee: - mtreinish - slowrie

Milestones

Target Milestone for completion:
Newton-2

References

- Newton Design Summit CLI Session
- Mitaka Design Summit CLI Session

Previous Implementations and Specs

- os-testr runner
- Prototype by mtreinish
- Previous Tempest CLI spec

```
This work is licensed under a Creative Commons Attribution 3.0 Unported␣
↪License.

http://creativecommons.org/licenses/by/3.0/legalcode
```

2.1.34 Test accounts

https://blueprints.launchpad.net/tempest/+spec/test-accounts

Tempest test accounts management

Problem description

Tempest relies on tenant isolation for parallel test executions. Test accounts are provisioned on the fly for each test class to ensure isolation. This approach requires an identity admin account being available for the provisioning. The aim of this blueprint is to provide an alternative solution, specifically pre-provisioned accounts, and to solve the problems related to this approach: how test accounts are allocated to different test processes spawned by testr, including alt user and other test accounts, and how we can support both tenant isolation and this new implementation in tempest, and easily switch between the two.

Proposed change

Abstract the existing tenant isolation mechanism to a credentials provider meta-class with two different implementations:

- A configured credentials provider, which uses credentials statically configured in tempest.conf. This can be used to avoid including admin credentials in the configuration at test run-time
- A dynamic credentials provider, which uses the existing tenant isolation logic to generate credentials on the fly using identity admin credentials

Move the logic that selects test accounts out of test and test base classes into a separate `credentials_factory` class, which provides the correct implementation of tenant isolation to tests based on configuration. The `credentials_factory` is the sole responsible for reading configuration related to test accounts, and the existing `get_credentials` methods from auth.py should be dropped, and the corresponding logic implemented in the configured credentials provider.

Change from something like this repeated in various slightly different flavours:

```
if CONF.compute.allow_tenant_isolation:
    cls.os = clients.Manager(cls.isolated_creds.get_primary_creds())
else:
    cls.os = clients.Manager()

if CONF.compute.allow_tenant_isolation:
    cls.os = clients.Manager(cls.isolated_creds.get_alt_creds())
else:
    cls.os_alt = clients.AltManager()
```

To a code where tenant isolation is hidden to tests:

```
# Provides the right implementation based on the configuration
cls.isolated_creds = credentials_factory.get_isolated_credentials()

# This may raise an AccountNotAvailable
cls.os = clients.Manager(cls.isolated_creds.get_primary_creds())
cls.os_alt = clients.Manager(cls.isolated_creds.get_alt_creds())
```

Preserve the tenant_isolation flag, but move it out of compute to a `common` configuration group. Remove the current user account settings from the identity section, and create lists of settings in the `common` configuration group instead, for users, tenants, passwords and domains.

As an indication users expects a list of values = CONCURRENCY x 2. Tenant, password and domain will expect a list of values either the same length as users, or with just 1 element, which represents the default value for all users.

Once a test is completed it releases the accounts it requested, and they are available for the next test to use. This should highlight issues with resource cleanup in tests, such as resource leaks and non-blocking deletes.

The configured credentials provider implements the logic to provide accounts to tests from the pre-configured ones, ensuring that one account is only used by one test only at any time. We use a file based reservation mechanism, which addresses the following issues:

- avoid the initial race where parallel test processes all want to allocate a test account at once
- ensure that file(s) used for reservation are cleanup up when testing is done, even though each process alone does not have the knowledge about when test overall is completed

How many accounts will tests require may vary depending on which tests are executed. At the moment two per process is typically enough, if we exclude identity tests, which we can do because they mostly require admin credentials anyways, so they would not run without admin credentials configured. Still it may happen that no account is available when test request it. In this case we the test will fail with AccountNotAvailable.

Assumptions

All non-admin test accounts have the same roles associated. All resources associated to an account that require admin credentials for creation are pre-created.

Alternatives

Implement static allocation of accounts to test processes to avoid the reservation system. That requires either naming conventions for test accounts, or each test process to be aware of its own ID, so that a selection can be made based on hashing.

Implementation

Assignee(s)

Andrea Frittoli <andrea.frittoli@hp.com>

Milestones

Target Milestone for completion:
Juno-final

Work Items

- Expose the tenant isolation interface into a metaclass
- Move the existing tenant isolation to the dynamic accounts provider
- Implement the static accounts provider and reservation mechanism, including the modified configuration options
- Implement a credentials_factory
- Adapt tests and test base classes (in multiple steps)

Dependencies

None

2.1.35 Test accounts Continued

https://blueprints.launchpad.net/tempest/+spec/test-accounts-continued

Tempest test accounts management

Problem description

The Test accounts spec provided support for preprovisioned accounts, as well as for those accounts to be configured in YAML format. There are a few limitations to the existing limitations:

- all accounts must belong to the same network
- all accounts must be of the same type, so we have a combination of accounts configured in tempest.conf and in accounts.yaml

Proposed change

Extend the format of the accounts YAML file to support specifying the name and type of resources pre-provisioned for an account. Such resources are intended to be reused by tests, and shall not be cleaned-up.

```
- credentials:
      username: 'user_1'
      tenant_name: 'test_tenant_1'
      password: 'test_password'
  resources:
      network: 'my_network'
      subnet: 'my_subnet'

- credentials:
      username: 'user_2'
      (...)
```

Extend the format of the accounts YAML file to support specifying the account type of an account. We may have an account type identifier, or alternatively a list of roles.

```
- credentials:
      username: 'admin'
      tenant_name: 'admin_tenant'
      password: 'admin_password'
  type: 'admin'

- credentials:
      username: 'swift_admin'
      tenant_name: 'admin_tenant'
      password: 'admin_password'
  roles:
      - reseller
```

Adapt the credentials providers to be able to handle credentials requests based on specific roles as well as account type (available today via dedicated methods).

The abstract implementation would be something like:

```
@abc.abstractmethod
def get_creds_by_roles(self, roles=None):
    return

def get_creds_by_type(self, type=None):
    if type == "primary":
        return get_primary_creds()
        (...)
```

Adapt the non pre-provisioned account scenario to also read accounts from the accounts YAML file, and deprecate any account information in tempest.conf beyond the name of account file.

Provide a tool to be consumed by devstack to generate the pre-provisioned accounts and the corresponding YAML file. Work on this is already started here https://review.openstack.org/#/c/107758/.

Integrate with the post-run clean-up tool, to avoid deleting pre-provisioned resources specified in the YAML file.

Alternatives

The current implementation is functional but incomplete, so the only alternative is not to use it, or suffer its limitation.

Implementation

Assignee(s)

Andrea Frittoli <andrea.frittoli@hp.com>

Milestones

Target Milestone for completion:
Kilo-final

Work Items

- Extend the YAML file parser
- Implement the non-cleanup of configure resources
- Deprecate account configuration options
- Read account info from YAML, with fallback to deprecated configuration options
- Implement provisioning tool
- Switch devstack and job definition to use the accounts YAML file in case of tenant isolation as well as pre-provisioned accounts
- Configure check/gate to run a combination of tenant isolation and pre-provisioned accounts

Dependencies

None

2.2 DevStack

2.2.1 Devstack external plugins

https://blueprints.launchpad.net/tempest/+spec/devstack-external-plugins

Support external plugins for devstack.

Problem description

Devstack has a pretty strong plugin support, you just have to drop your extra feature in the extras.d/ directory and it will automatically parse the file and install the feature as long you have enabled it in your local.conf.

This all works very well but thats not very flexible for projects that are not able to be integrated directly in devstack core. Currently an external OpenStack project who wants to tell its users how to test a feature has to explain how to download a file to put in the *extras.d* directory and enabling it.

As for integrated projects they may wants to take care of how they do devstack directly in their own repo and get devstack to use that instead of having to request for a change in devstack repository.

Proposed change

Devstack would provide a new enable_plugin function call that would be of the following format:

```
enable_plugin <name> <http://git.openstack.org/foo/external_feature>
[refname]
```

name is an arbitrary name picked for enablement, *repo* is the full url to a git repo, and *refname* is the optional ref description (defaulting to *master* if none is provided).

Devstack would then checkout that repository in *${DEST}/name* and look for a */devstack/* directory in there from the root of the repo.

Files in there would have :

- */devstack/settings* - a file that gets sourced to override global settings, those variables become instantly available in the global namespace.
- */devstack/plugin.sh* - dispatcher for the various phases

Devstack when executed will then:

- Get the configuration of the *extras.d* repos.
- Clone the extras repository to *${DEST}*
- Run all the *extras.d* scripts at a particular phase
- Run all the plugins *plugin.sh* at a particular phase

This would let the out of tree projects that needs to communicate about their config to export data via settings that would let the other configure based on their setup.

Alternatives

The alternative would be to stay as the status quo like we have now and have them to curl the extras file from the external repository and place it in the *extras.d* directory.

Implementation

Assignee(s)

Primary assignee:

- Chmouel Boudjnah <chmouel@chmouel.com>

Other contributors:

- Sean Dague <sean@dague.net>

Milestones

Target Milestone for completion:
Kilo-2

Work Items

- Add support in devstack.
- Get an example repository setup.
- Get a project like nova-docker to use it. * (glusterfs is a good current candidate)

2.2.2 DevStack Logging and Service Names

NOTE: This spec is still a work in progress, it is being posted to get some early feedback on scope and ordering of steps.

https://blueprints.launchpad.net/tempest/+spec/devstack-logging-and-service-names

DevStack is in need of updates to its log file handling and service naming, both of which were appropriate for the originalscreen-based installs with a handful of services.

This spec contains both the log file reform as well as the service name updates as they are a bit intertwined so some steps will address them both as necessary.

Problem description

DevStacks logging configuration was initially based on saving `screen` logs, as part of the development of not using `screen` the logging was kept compatible and it became obvious that the original special case was not required.

Historically DevStack has used abbreviated service names for identifying services to enable, naming log files and as window names in screen. OpenStack has grown to the point that the abbreviated names are too confusing and non-obvious, especially for the not-so-recently renamed Neutron.

These topics were covered at the Paris summit, notes in the OpenStack Etherpad.

Proposed change

Logging

Update DevStacks logging configuration to set a logging directory rather than parsing that out of a filename. Ultimately eliminate the use of `SCREEN_LOGDIR`.

- Use `LOGDIR` as the primary setting in local.conf for log locations, default to `${DEST}/logs` if `LOGFILENAME` is not set.
- Continue to use `LOGFILENAME` if set, if `LOGFILE` is not set continue to set it to $(dirname `$LOGFILENAME`).
- Deprecate `SCREEN_LOGDIR` and use `LOGDIR` instead. For a compatibility period leave symlinks in the old screen log locations.
- Remove `screen-` from the beginning of the service log filenames
- Service log files will implicitly be renamed as the service names change (see above)

Grenade should work seamlessly as it lets both DevStack runs do their thing and `devstack-gate` contains all of the specifics that need updating fro Grenade jobs.

Service Names

Use fully-formed names for service names (like ceilometer does today): `nova-compute`, `glance-registry`, etc. The names will use the project name, as used in `devstack/lib/*` followed by - and a descriptive name of the service.

Also allow multiple instances of service names, as in running the fake hypervisor has a number of `nova-cpu` instances. Append an instance counter to the name similar to how `n-cpu-N` is currently handled. Optionally use a : as the separator between the service name and instance number. This will be used in the log file name so it must be shell-safe.

- There needs to be a mapping of the old abbreviated names to the full names to handle backward compatibility.
- This will make `ENABLED_SERVICES` very long by default and harder to scan visually. Is this a real concern? With the recent forced update to using Bash 4 we could use an associative array to do the mapping and the enabled list in a single shot.

  (Note: We just started doing something in Grenade to handle mapping abbreviated service names-> processes (https://review.openstack.org/#/c/113405/5/check-sanity) This would help move that logic into DevStack and also help provide other mappings (ie, service name -> database name))
- Log filenames will change, but there is more on that front (see below).
- Grenade will need to be updated before the backward compatibility can be removed.

Implementation

Assignee(s)

Primary assignee:
dtroyer

Work Items

1. Logging: change the log file names in `SCREEN_LOGDIR` so the actual files with the timestamp in the names end with the timestamp:

```
screen-c-sch.2014-12-10-193405.log becomes  screen-c-sch.log.2014-12-10-
↪193405
```

2. Logging: change `devstack-gate` to look for `*.log` rather than symlinks to select the log files it copies out of `screen-logs`.
3. Logging: switch from `SCREEN_LOGDIR` to `LOGDIR` for log tests. This will move the log files out of `SCREEN_LOGDIR` so leave backward-compatibility symlinks in the old locations. (This is the reason for #2 as `devstack-gate` selects the files top copy by the symlink attribute.)
4. Logging: follow up in `devstack-gate` to use LOGDIR directly and copy log files from there.
5. Logging: after a time, remove the symlinks from `SCREEN_LOGDIR`.
6. Services: change how multiple instances of services are handled, currently in `lib/nova` `start_nova_compute()` and `stop_nova_compute()`. If the separator is changed the config filenames will also change, reconsider if parsing is necessary.
7. Services: build the new service naming structures and compatibility.
8. Services and Logging: switch logging to use the new service names and ensure nothing gets lost in `devstack-gate` copies.

Dependencies

The only dependencies are in the order of changes required in multiple projects.

2.3 Patrole

2.3.1 RBAC Policy Testing

https://blueprints.launchpad.net/tempest/+spec/rbac-policy-testing

OpenStack deployments have standard RBAC (Role-Based Access Control) policies that are customized in different ways by users of OpenStack. These policies need to be enforced and verified to ensure secure operation of an Openstack Deployment.

Problem Description

Currently there is no unified way to test that RBAC policies are correctly enforced. This is important because these policies define how potentially sensitive information and functionality are accessed.

Proposed Change

The proposed solution involves creating a plugin that enables RBAC tests to be written in such a way that they can verify that an individual policy was correctly enforced on the specified rbac_role. Tests should primarily be written as an extension of an existing Tempest test that is wrapped in the plugins functionality. The plugin determines what roles should have access to a given API based on the policy.json file for that service.

For example, a test can be written that only tests the policies for compute_extension:services as follows:

```
@test.requires_ext(extension='os-services', service='compute')
@rbac_rule_validation.action(
    component="Compute",
    rule="compute_extension:services")
@test.idempotent_id('ec55d455-bab2-4c36-b282-ae3af0efe287')
def test_services_ext(self):
    try:
        rbac_utils.switch_role(self, switchToRbacRole=True)
        self.client.list_services()
    finally:
        rbac_utils.switch_role(self, switchToRbacRole=False)
```

A key aspect of these tests is that they are testing only a single policy, even though a normal flow might require the usage of actions covered by multiple policies. To enable this, the role used for the tests will alternate between admin and the rbac_role specified in the config options. This is done so that there is no chance of other policies besides the one being tested changing the outcome of the test. To switch roles, the switch_role method of rbac_utils is called. Calling the method with switchToRbacRole=True tells Keystone to set the current role to the rbac_role while switchToRbacRole=False tells Keystone to set the current role to admin.

This effort involves creating the plugin for new tests that verify correct RBAC policy enforcement. This effort will include sample tests but is not intended to provide full testing coverage of all policies and APIs. The core code will exist in an external repository containing a plugin, while individual tests will be written within the plugin. There is not currently any functionality that needs to be added to tempest/lib, but that may change as more tests are written.

Alternatives

An alternative is to utilize the cinnamon-role plugin that was being considered.

Advantages: Utilizing the cinnamon-role plugin enables RBAC functionality to be wrapped around existing tests with minimal extra effort.

Disadvantages: Cinnamon-role doesnt allow for testing of individual API endpoints by switching role mid-test.

Another alternative is to add the RBAC test framework into Tempests core functionality.

Advantages: Easier deployment, dont need extra addons.

Disadvantages: Rbac testing is not something that is considered part of Tempests core functionality.

Projects

List the qa projects that this spec effects. For example: * openstack/tempest

Implementation

Assignee(s)

Primary assignees:
david-purcell [david.purcell@att.com] jallirs [randeep.jalli@att.com] syjulian [julian.sy@att.com] fm577c [felipe.monteiro@att.com] sblanco1 [samantha.blanco@att.com]

Milestones

Target Milestone for completion:
ocata-2

Work Items

- create plugin for role testing
- add supporting code to tempest/lib/rbac if needed
- add initial group of tests

Dependencies

None

2.3.2 RBAC Testing Multiple Policies

bp rbac-testing-multiple-policies

Problem Description

Patrole currently RBAC tests an API endpoint by checking whether a policy action is allowed, according to `oslo.policy` and then executes the API endpoint that does policy enforcement with the role specified under `CONF.rbac.rbac_test_role`. However, this approach does not account for API endpoints that enforce multiple policy actions, either directly (within the implementation of the API endpoint itself) or indirectly (across different helper functions and API endpoints). The current approach to RBAC testing in Patrole, therefore, does not always provide complete policy coverage. Just like multiple calls are made to `oslo.policy` by various endpoints, Patrole should do the same.

For example, take an API that enforces 2 policy actions, A and B, where A is `admin_api` and B is `admin_or_owner`. Calling the API with `rbac_test_role` as admin role will necessarily pass, because admin role has permissions to execute policy actions A and B and will also be able to execute the API endpoint. However, the test will fail for non-admin role, with the `rbac_rule_validation` decorator evaluating *only* policy action B. This is because a non-admin role (i.e. Member) role *has* permissions to perform policy action B (which is `admin_or_owner`) but does *not* have permissions to execute the API endpoint, since the endpoint enforces an `admin_api` policy: this results in a `Forbidden` exception being raised, and the test failing.

Proposed Change

The proposed change is to modify the `rbac_rule_validation` decorator to be able to take a list of policy actions, rather than just one policy action. For each policy action, a call will be made to `oslo.policy` to confirm whether the test role is allowed to perform the action. Each result from `oslo.policy` will be logical-ANDed together. For example, if policy action A evaluates to `True` and policy action B evaluates to `False`, then the final outcome is `False`: therefore, the user should not be able to perform the API call successfully. As such, Patrole can deduce whether a role is allowed to call an API that enforces multiple policies.

To provide a concrete example, the following test:

```
@rbac_rule_validation.action(
    service="nova",
    rule="os_compute_api:os-lock-server:unlock:unlock_override")
def test_unlock_server_override(self):
    server = self.create_test_server(wait_until='ACTIVE')
    # In order to trigger the unlock:unlock_override policy instead
    # of the unlock policy, the server must be locked by a different
    # user than the one who is attempting to unlock it.
    self.os_admin.servers_client.lock_server(server['id'])
    self.addCleanup(self.servers_client.unlock_server, server['id'])

    self.rbac_utils.switch_role(self, toggle_rbac_role=True)
    self.servers_client.unlock_server(server['id'])
```

can be changed to:

```
@rbac_rule_validation.action(
    service="nova",
    rules=["os_compute_api:os-lock-server:unlock",
           "os_compute_api:os-lock-server:unlock:unlock_override"])
def test_unlock_server_override(self):
    server = self.create_test_server(wait_until='ACTIVE')
    self.os_admin.servers_client.lock_server(server['id'])
    self.addCleanup(self.servers_client.unlock_server, server['id'])

    self.rbac_utils.switch_role(self, toggle_rbac_role=True)
    self.servers_client.unlock_server(server['id'])
```

According to the Nova documentation for locking a server, the unlock_override policy is performed only after the check os_compute_api:os-lock-server:unlock passes. With this change, Patrole will generate its expected result based on whether the test role can perform *all* the policies passed to `rules`; otherwise,

if the test role cannot perform at least one policy, the expected result will be `False`. Afterward, the API action will be called with the test role and the outcome of which will be compared with the expected result.

If the expected and actual results match, then the test will pass. Otherwise, Patrole can generate a detailed error message explaining which policies passed to `rules` caused test failure. For example, in the above example, if the test role has permissions to perform os_compute_api:os-lock-server:unlock but not os_compute_api:os-lock-server:unlock:unlock_override, then Patrole will emit an error saying that os_compute_api:os-lock-server:unlock:unlock_override was responsible for test failure. This will help cloud deployers and developers to determine the source of test failure and to pinpoint inconsistent custom policy configurations.

Alternatives

Currently, there are no other viable alternatives. It is not feasible or desirable to repeatedly call each API endpoint against each policy action that the endpoint enforces, for various fairly obvious reasons:

1. Code redundancy should be minimized, to make code readability and maintenance easier.
2. This introduces serious run time concerns in Patroles gates.

Security Impact

None.

Notifications Impact

`LOG` statements will have to be updated to convey multiple policy actions to the user, especially following test failure.

If a Patrole test tests may policies, after test failure, it would be useful for users for Patrole to log which policies caused the test failure. This can be determined by iteratively calling `oslo.policy` for each policy provided to the `rbac_rule_validation` decorator and storing the list of policies that are not compatible with the role and the expected test outcome.

Other End User Impact

None.

Performance Impact

The performance impact is negligible. This change will result in barely slower test run time, because multiple calls will be made to `oslo.policy` rather than just one, per Patrole test.

Other Deployer Impact

None.

Developer Impact

The proposed change requires that developers be *prudent* about which policy actions they include in the proposed `actions` parameter. Including an excessively high number of policy actions is not maintainable and is cumbersome from a development standpoint. For example, Cinder enforces `volume_extension:volume_host_attribute` and `volume_extension:volume_mig_status_attribute`, along with a number of different policy actions, for many, many endpoints. Repeating these policy actions for every Cinder RBAC test would be redundant and bad design. (If it could be proven that these policy actions are enforced for *every* Cinder API endpoint, then the policy actions could be auto-injected by the Patrole framework and logical-ANDed with the policy actions explicitly specified in `actions`. However, this approach goes beyond the scope of this spec).

It is recommended that this enhancement be used *judiciously* by developers. Only endpoints that enforce multiple relatively *unique* policy actions should be included in the `actions` list. Uniqueness can be inferred, for example, from Keystones and Novas self-documenting in-code policy definitions.

Implementation

Assignee(s)

Primary assignees:

- Felipe Monteiro <felipe.monteiro@att.com>
- Samantha Blanco <samantha.blanco@att.com>

Other contributors:

- Rick Bartra <rb560u@att.com>

Work Items

- Enhance the `rbac_rule_validation` decorator with the `actions` parameter and deprecate the `rule` parameter.
- Write a helper function in `rbac_rule_validation` to iteratively call `rbac_policy_parser.RbacPolicyParser.allowed` for each policy action specified in `actions`, logically ANDing them together, and returning the result to `rbac_rule_validation` decorator.
- Refactoring tests to use `actions` instead of `rule`.
- Writing new unit tests to test the proposed enhancement.
- Selectively adding multiple policy actions to some tests.
- Confirming that all API tests work with the proposed enhancement.
- Updating documentation.

Dependencies

None.

Documentation Impact

Patrole documentation should be updated to convey the new parameter along with intended use, as described in this spec.

References

- Policy terminology
- Keystone policy in code
- Nova policy in code

2.4 Other

2.4.1 Placeholder

This is just a placeholder file to avoid sphinx errors. Please remove this file when you add a new rst file.

CHAPTER

THREE

SPECIFICATION REPOSITORY INFORMATION

3.1 Team and repository tags

3.2 QA Specs Repository

This is a git repository for doing design review on QA enhancements as part of the OpenStack program. This provides an ability to ensure that everyone has signed off on the approach to solving a problem early on.

This repository includes the Tempest and DevStack projects.

3.2.1 Repository Structure

The structure of the repository is as follows:

```
specs/
    devstack/
        implemented/
    other
        implemented/
    tempest
        implemented/
```

3.2.2 Expected Work Flow

1. Create a blueprint stub in `tempest`, `devstack`, or `other` blueprint repository
2. Propose review to qa-specs repository (ensure bp:blueprint_name is in the commit message. DevStack specs should go into the `devstack/` subdirectory but otherwise follow the same process.
3. Link `Read the full specification` to the gerrit address of the spec
4. Bring forward the proposed item to the openstack-qa meeting for summary
5. Review happens on proposal by qa-core members and others
6. Iterate until review is Approved or Rejected

Once a Review is Approved

1. Update blueprint, Copy summary text of blueprint to there
2. Link `Read the full specification` to the git address of the spec
3. Profit!

3.2.3 Revisiting Specs

We dont always get everything right the first time. If we realize we need to revisit a specification because something changed, either we now know more, or a new idea came in which we should embrace, well manage this by proposing an update to the spec in question.

3.2.4 Learn as we go

This is a new way of attempting things, so were going to be low in process to begin with to figure out where we go from here. Expect some early flexibility in evolving this effort over time.

3.2.5 Tempest Specs For New Tests

If youre writing a new spec to improve the testing coverage in Tempest the requirements for what is included in the specification are slightly less stringent and different from other proposals. This is because blueprints for more tests are more about tracking the effort in a single place and assigning a unified topic in gerrit for ease of review, its less about the implementation details. Blueprints/specifications for new tests should only ever be opened for overarching development efforts. For example there should only ever only need to be a single blueprint for adding tests for a project.

Most of these efforts require a method to track the work items outside of launchpad. Both etherpad and google docs have been used very successfully for this. The goal is to list out all the tests that need to be written and allow people to mark that they intend to work on a specific test. This prevents duplication of effort as well as provide overall status tracking. An external tool like etherpad or google docs is better at this because it allows concurrent use and more dynamic editing than launchpad.

The only details required in the proposed change section for a spec about new tests are:

- What is being tested and the scope of what will be covered by the blueprint
- What external tool is being used to track the development.
  - If no external tracking is being used just explain why.

3.2.6 DevStack Specs

Specs for DevStack fall into a couple of broad categories:

- Support for new {project|driver|cool widget}

  This is where the discussion of Does this support belong in the DevStack repo? should take place.

- Significant re-factoring

  One primary section that these types of changes require is an analysis of backward compatibility and Grenade impacts.

The existing template is mostly suitable for DevStack use, a quick s/tempest/devstack/ handles the majority of changes.

CHAPTER

FOUR

INDICES AND TABLES

- search