Fuel Noop Tests Documentation

Release 0.1

Mirantis inc

Mar 22, 2017

Contents

CHAPTER 1

Abstract

The fuel-library is collection of Puppet modules and related code used by Fuel to deploy OpenStack environments. There are top-scope Puppet manifests, known as a Fuel library modular tasks. This guide documents the Fuel Noop testing framework for these modular tasks.

CHAPTER 2

Contents

Fuel Noop fixtures

There is a separate fuel-noop-fixtures repository to store all of the fixtures and libraries required for the noop tests execution. This repository will be automatically fetched before the noop tests are run to the *tests/noop/fuel-noop-fixtures* directory.

Developers of the noop tests can add new Hiera and facts yaml files into this repository instead of the main fuel-library repository starting from the Fuel Mitaka (9.0) release.

Note: The fixtures for the Fuel <=8.0 belong to the fuel-library repository and must be changed there.

Automatic generation of fixtures

The fixtures must contain data as it comes from the Fuel deployment data backend (Nailgun). Fixtures contain only data specific to the corresponding Fuel version. Manual changes to the fixtures' data should be avoided.

The current approach to generate the fixtures is a semi-automated and requires a Fuel master node of a given release deployed. To generate the fixtures, for each of the deployment cases (environments) under test, first create the environment, for example:

```
$ fuel env --create --name test_neutron_vlan --rel 2 --net vlan
```

Then query, update and upload the environment attributes as required. For example, to test a Ceph-for-all-but-ephemeral-plus-Ceilometer deployment:

```
$ fuel env --attributes --env 1 --download
$ ruby -ryaml -e '\
> attr = YAML.load(File.read("./cluster_1/attributes.yaml"))
> attr["editable"]["storage"]["images_ceph"]["value"] = true
> attr["editable"]["storage"]["objects_ceph"]["value"] = true
> attr["editable"]["storage"]["volumes_ceph"]["value"] = true
> attr["editable"]["storage"]["volumes_lvm"]["value"] = false
> attr["editable"]["additional_components"]["ceilometer"]["value"] = true
> File.open("./cluster_1/attributes.yaml", "w").write(attr.to_yaml)'
$ fuel env --attributes --env 1 --upload
```

At last, add nodes, assign roles as you want to test it, then generate and store the data fixtures as YAML files, for example:

```
$ fuel --env 1 node set --node 1 --role controller
$ fuel --env 1 node set --node 2 --role compute,ceph-osd
$ fuel deployment --default --env 1
$ ls /root/deployment_1
ceph-osd_2.yaml  compute_2.yaml  primary-controller_1.yaml
```

Those files are now ready to be renamed and put under the *hiera* directory, like this:

```
$ git clone https://github.com/openstack/fuel-noop-fixtures
$ mv /root/deployment_1/compute_2.yaml \
> ./fuel-noop-fixtures/hiera/neut_vlan.ceph.ceil-compute.yaml
$ mv /root/deployment_1/ceph-osd_2.yaml \
> ./fuel-noop-fixtures/hiera/neut_vlan.ceph.ceil-ceph-osd.yaml
$ mv /root/deployment_1/primary-controller_1.yaml \
> ./fuel-noop-fixtures/hiera/neut_vlan.ceph.ceil-primary-controller.yaml
```

Note, there is a script to automate things to a certain degree as well. Hopefully, we will improve the auto-generation process, eventually.

Structure

Data files

To run a noop test on a spec following files are required:

- A spec file: *(i.e. spec/hosts/my/my_spec.rb)*
- A task file: *(i.e. modular/my/my.pp)*
- One of the Facts sets: *(i.e. ubuntu.yaml)*
- One of the Hiera files: *(i.e. neut_vlan.ceph.controller-ephemeral-ceph.yaml)*

Any single task is a combination of three attributes: spec file, yaml file and facts file. Manifest file name and location will be determined automatically based on the spec file. RSpec framework will try to compile the Puppet catalog using the manifest file and modules from the module path. It will use the facts from the facts file and the Hiera data from the hiera file.

If the spec is empty it will test only that catalog have compiled without any errors. It's actually not a bad thing because even empty specs can catch most of basic errors and problems. But if the spec has a **shared_examples 'catalog'** block defined and there are several examples present they will be run against the compiled catalog and the matchers will be used to determine if examples pass or not.

Every Hiera yaml file also has a corresponding *globals* yaml file that contains additional processed variables. These files are also used by most of the spec tests. If you make any changes to the hiera yaml files you should also recreate globals files by running *globals/globals* specs with *save globals* option enabled. Later new files can be commited into the fixtures repository.

And, finally, there is an override system for hiera and facts yaml files. For each spec file you can create a hiera or facts yaml file with a special name. This file will be used on top of other files hierarchy. It can be very useful in cases when you need to provide some custom data which is relevant only for the one task you are working with without touching any other tasks.

Framework components

The Noop test framework consists of the three components: the task manager, the config and the task.

The task manager is responsible for collecting the information about the present files, manipulation the task library, processing the console options and environment variables and, finally, running the tasks using the tasks objects and processing reports.

The config object contains the basic information about directory structure and some default values and the values passed fro the external environment variables. This object is static and is persistent between the instances of all other objects.

The task object is the instance of a single test run. It can work with spec, manifest, Hiera and facts yaml paths and run the actual RSpec command to start the test.

The similar instance of the task process will be created inside the RSpec process namespace and will be used to provide the information about the current task as well as providing many different helpers and features for the spec users. This object can be accessed through the proxy method of the root **Noop** object which keep the reference to the current task instance.

Using the noop_tests utility

The noop_tests options

Noop tests framework is actually located in the fixtures repository together with its yaml data files. There is a wrapper script *tests/noop/noop_tests.sh* that can be used from the Fuel library repository to automatically setup the external fixtures repository, configure paths and run the framework.

First, you can use the **-h** options to get the help output.:

```
tests/noop/noop_tests.sh -h
```

Output::

```
Usage: noop_tests [options]
Main options:
    -j, --jobs JOBS                  Parallel run RSpec jobs
    -g, --globals                    Run all globals tasks and update saved globals␣
↪YAML files
    -B, --bundle_setup               Setup Ruby environment using Bundle
    -b, --bundle_exec                Use "bundle exec" to run rspec
    -l, --update-librarian           Run librarian-puppet update in the deployment␣
↪directory prior to testing
    -L, --reset-librarian            Reset puppet modules to librarian versions in␣
↪the deployment directory prior to testing
    -o, --report_only_failed         Show only failed tasks and examples in the report
    -O, --report_only_tasks          Show only tasks, skip individual examples
    -r, --load_saved_reports         Read saved report JSON files from the previous␣
↪run and show tasks report
    -R, --run_failed_tasks           Run the task that have previously failed again
    -x, --xunit_report              Save report in xUnit format to a file
List options:
    -Y, --list_hiera                List all hiera yaml files
    -S, --list_specs                List all task spec files
    -F, --list_facts                List all facts yaml files
    -T, --list_tasks                List all task manifest files
Filter options:
    -s, --specs SPEC1,SPEC2         Run only these spec files. Example: "hosts/hosts_
↪spec.rb,apache/apache_spec.rb"
    -y, --yamls YAML1,YAML2         Run only these hiera yamls. Example: "controller.
↪yaml,compute.yaml"
    -f, --facts FACTS1,FACTS2       Run only these facts yamls. Example: "ubuntu.
↪yaml,centos.yaml"
Debug options:
    -c, --task_console              Run PRY console
    -C, --rspec_console             Run PRY console in the RSpec process
    -d, --task_debug                Show framework debug messages
    -D, --puppet_debug              Show Puppet debug messages
        --debug_log FILE            Write all debug messages to this files
    -t, --self-check                Perform self-check and diagnostic procedures
    -p, --pretend                   Show which tasks will be run without actually␣
↪running them
Path options:
        --dir_root DIR              Path to the test root folder
        --dir_deployment DIR        Path to the test deployment folder
        --dir_hiera_yamls DIR       Path to the folder with hiera files
        --dir_facts_yamls DIR       Path to the folder with facts yaml files
        --dir_spec_files DIR        Path to the folder with task spec files␣
↪(changing this may break puppet-rspec)
        --dir_task_files DIR        Path to the folder with task manifest files
        --dir_puppet_modules DIR    Path to the puppet modules
Spec options:
    -A, --catalog_show              Show catalog content debug output
    -V, --catalog_save              Save catalog to the files instead of comparing␣
↪them with the current catalogs
    -v, --catalog_check             Check the saved catalog against the current one
    -a, --spec_status               Show spec status blocks
        --puppet_binary_files       Check if Puppet installs binary files
        --save_file_resources       Save file resources list to a report file
```

Shortcut scripts

There are also several shortcut scripts near the *noop_tests.sh* file that can be used to perform some common actions.

- **tests/noop/noop_tests.sh** The main wrapper shell script. It downloads the fixtures repository, sets the correct paths and setups the Ruby gems. It's used by many other shortcut scripts.
- **utils/jenkins/fuel_noop_tests.sh** The wrapper script used as an entry point for the automated Jenkins CI jobs. Runs all tests in parallel mode.
- **tests/noop/run_all.sh** This wrapper will run all tests in parallel mode.
- **tests/noop/run_global.sh** This wrapper will run all globals tasks and save the generated globals yaml files.
- **tests/noop/setup_and_diagnostics.sh** This wrapper will first setup the Ruby environment, download Fuel Library modules and run the noop tests in the diagnostics mode to check the presence of all folders in the structure and the numbers of tasks in the library.

- **run_failed_tasks.sh** This wrapper will load the saved reports files from the previous run and will try to run all the failed tasks again.
- **purge_reports.sh** Removes all task report files.
- **purge_globals.sh** Removes all saved globals files.
- **purge_catalogs.sh** Removes all saves catalog files.

Typical use cases

Let's discuss the most common use cases of the Noop test framework and how it can be used.

Initial setup

In most cases you should setup the environment before using the Noop tests framework. The setup consists of three parts:

- Fixtures repository clone
- Ruby gems installation
- Puppet modules download

There is a wrapper script **tests/noop/setup_and_diagnostics.sh** that will try to do all these things. First, it will clone the fixtures repository unless it have already been cloned, then it will run **tests/noop/noop_tests.sh** with options **-b** and **-B** to create and use the bundle gems folder, **-l** options will enable Puppet modules update and **-t** option will initiate check procedures for paths and task library content.

If you are using *RVM* or are managing Ruby gems manually you are free to bypass this stem and clone fixtures repository manually.

Running all tests using multiple processes

Running **tests/noop/noop_tests.sh** without any options will try to execute all the spec tasks one by one. The list of the spec tasks will be generated by combining all the possible combinations of specs, hiera files and facts files that are allowed for this spec. Adding **-p** options allows you to review the list of tasks that will be run without actually running them.

Running tasks one by one will be a very time consuming process, so you should try to use multi-process run instead by providing the **-j** options with a number of processes that can be started simultaneously. In this mode you will not see the output of every RSpec process, but you will be able to get the combined result report of all test runs at the end of the process.

You can also monitor the progress of tasks by using the debug option **-d**. It will show you which tasks are starting and finishing and what shell commands and environment variables are used to run them.:

```
tests/noop/noop_tests.sh -j 24 -d
```

Will run all the spec task with debug output and keeping no more then 24 child processes all the time.:

```
tests/noop/noop_tests.sh -p
```

Will output the list of tasks that is going to be run together with the facts and yaml files used.

There is also the **run_all.sh** shortcut script for this action.

Running only a subset of tasks

In many cases you would want to run only a subset of tasks, up to only one task. You can use filters to do so. By providing the **-s**, **-y** and **-f** will allow you to set one or more specs, yams and facts that you want to use. The list of tasks will be build by filtering out everything else that you have not specified. Don't forget that you can use the **-p** option to review the list of tasks before actually running them.

List options **-Y**, **-F**, **-S** and **-T** can be used to view the list of all hiera yaml files, facts files, spec files and task files respectively. These lists are very helpful for finding out correct values for the filter options you want to use. Note, that using filter and list options together will allow you to see the filtered lists.:

```
tests/noop/noop_tests.sh -Y
```

Will list all available hiera yaml files.:

```
tests/noop/noop_tests.sh -y neut_vlan.compute.ssl.yaml -p
```

Will show you which task are going to run with this yaml file.:

```
tests/noop/noop_tests.sh -y neut_vlan.compute.ssl -s firewall/firewall -f ubuntu
```

Will run the *firewall/firewall* spec test with the provided yaml and facts file, but it will only work if this combination is allowed for this spec. Note, that you can either provide **.yaml**, **_spec.rb**, **.pp** extensions for yaml and spec files, or you can omit them and they will be found out on their own.:

```
tests/noop/noop_tests.sh -y neut_vlan.compute.ssl,neut_vlan.compute.nossl -s firewall/
↪firewall,netconfig/netconfig -p
```

Filters can use used with a list of elements or can be given as a regular expression or a list of regular expressions:

```
./noop_tests.sh -p -s 'master/.*'
```

Will filter all tasks in the *master* group.:

```
./noop_tests.sh -p -s '^ceph.*,^heat.*,glance/db_spec'
```

Will filter all *ceph*, *heat* tasks and glance/db_spec task individually.:

```
./noop_tests.sh -p -s '^ceph.*' -y ceph
```

All *ceph* related tasks only on Hiera files which have Ceph enabled.

Recreating globals yaml files

All globals files should already be precreated and commited to the fixtures repository and there is no need for you to create them again in the most cases. But, if you have made some changes to the existing yaml files or have added a new one, you should create the globals yamls again.

You can do it by running *tests/noop/noop_tests.sh* with **-g** option. It will set filters to run only the globals tasks as well as enabling the option to save the generated yaml files. Using **-j** option will make the process much faster.

There is also the **run_globals.sh** shortcut script for this action.

Spec file annotations

The framework builds the list of tasks to run by combining all allowed facts and yaml files for each spec file and creating a task for every combination.

By default, the list of yaml files, allowed for each spec will be determined by the intersection of node roles the spec should be run on (obtained from the *tasks.yaml* files used by the Nailgun to compile the deployment graph) and the hiera file roles (obtained from the hiera files themselves). The facts file will default to **ubuntu** value.

In most cases it would be better to manually specify the hiera files and, possibly, the facts files for your spec file, because running a task for every hiera file with the same roles would be redundant. On the other hand, if you know which Hiera files can cause a different behaviour of your task, and you want to test this behaviour in the different scenarios, you can explicitly specify the list of yaml files and facts files you need.

The list of Hiera files can be set by using the **HIERA:** commented annotation string followed by the list of hiera file names separated by the space character.:

```
# HIERA: neut_vlan.compute.ssl neut_vlan.compute.nossl
```

The list of facts files can be specified the same way using the **FACTS:** annotation.:

```
# FACTS: centos6 centos7
```

The list of task will contain this spec with all possible combinations of the specified Hiera and facts files. If you need to enter only the precise list of possible run combinations you can use the **RUN:** annotation.:

```
# RUN: (hiera1) (facts1)
# RUN: (hiera2) (facts2)
```

It can be specified many times an all entered combinations will be added to the list.

You can use **ROLE** annotation to specify the list of node roles this spec should be running at. It will find the list of Hiera yaml files that have roles matching this list.:

```
# ROLE: controller
# ROLE: primary-controller
```

There is also a way to use the reverse logic. You can specify the Hiera and facts yaml files that you want to exclude from the list instead of providing the list of included files.:

```
# SKIP_HIERA: neut_vlan.compute.ssl neut_vlan.compute.nossl
# SKIP_FACTS: centos6
```

These yaml files will be excluded from the list of possible yaml files. If you have used both include and exclude options, the exclude option will have the priority over the include option. If there are no included Hiera files the list of Hiera files will be generated from the node roles.

Note, that all these options can be combined all together. It will mean to take all 'compute' yamls, add neut_vlan_l3ha.ceph.ceil-primary-controller and remove 'neut_vlan.compute.ssl' and then add master/master_centos7 run.:

```
# ROLE: compute
# HIERA: neut_vlan_l3ha.ceph.ceil-primary-controller.yaml
# RUN: master master_centos7
# SKIP_HIERA: neut_vlan.compute.ssl.yaml
```

The final annotation **DISABLE_SPEC** allows you to temporarily disable the spec from being seen the framework. It can use useful if you want to turn off a spec with run problems and fix them later without breaking the tests.:

```
# DISABLE_SPEC
```

The spec file with this annotation will be completely ignored.

Using hiera and facts overrides

In some cases you need a special set of facts values or the Hiera data for your task. If this values are very specific and are not useful for other tasks you can use override system instead of creating the new Hiera or facts yaml.

There are *override* folders inside the Hiera and facts folders. If you place a yaml file with the specific name to this folder, it will be used during the spec catalog compilation as the top level of Hiera's hierarchy. The values which are specified there will be used before the values in other yaml files. Hash values will be merged on the basic values and the matching key will be rewritten. Facts yamls work the same way by rewriting the basic values by the values specified in the override file.

Both yaml files should be named after the task name with path separator changed to the dash character. For example, the **firewall/firewall** task will use the override file name *firewall-firewall.yaml* and **openstack-controller/keystone** task will use the file name *openstack-controller-keystone.yaml* if these files are found in the override folders.

Using hiera plugin overrides

If you have several additional YAML files that should be applied on top of the base Hiera files, for example, files, provided or generated by plugins or the other tasks, you can use the plugin override system. Any files which have been placed to the *hiera/plugins/${yaml_base_name}/* folder will be applied on top of this YAML file when it will be used in the Noop tests.

Multiple files inside this directory will be ordered alphabetically with the former letters having the higher priority.

Working with report files

When the task manager runs the tasks they leave report files anf the manager can collect them to generate a combined test report seen at the end of the test process. These files can be found in the reports folder and re in json format.

You can use **-r** and **-R** options to load the saved reports from the previous run and display the report again, or to load reports and run the tasks that have previously failed after you have tried to somehow fix them.

You can use option **-o** to filter out only failed tasks and examples from the report and **-O** options to show only tasks without showing the individual examples. These options can be used together to show only failed tasks.

The task manager can also generate a test report in *jUnit XML* format using the **-x** options. It will be saves to the **report.xml** file in the *reports* folder of the fixtures repository. This file can be used by many tools to visualize the tests results, notably by the Jenkins CI.

Catalog debugging

There are several features that can be helpful during writing the initial spec for a task or when you are debugging a spec failure. Running tasks with **-a** options will show the report text about which files are being used in this task run and what files are found in Hiera and facts hierarchies.

Using **-A** option will output the entire compiled catalog in the Puppet DSL format. You can review its content and resource parameters to either find out what resources and classes are there or to see what values the parameters and properties actually have after all the catalog logic is processed. It's very helpful when you are debugging a strange task behaviour or writing a spec file.

The framework can also gather and report information about *File* resources that are being installed by Puppet. Using *–save_file_resources* options will dave the list of files that would be installed by the catalog and description about their source or template. Using *–puppet_binary_files* option will enable additional RSpec matcher that will fail if there are files and, especially, binary files being installed. These ones should be delivered by fuel packages.

Data-driven catalog tests

Usually the spec files try to repeat the logic found in the tested manifests, receive the same set of resources and their parameters and compare them to the set of resources found in the compiled catalog. Then the matchers are used to check if the catalog contains what is expected from it to contain.

While this method works well in most cases it requires a lot of work and extensive expertise in the tasks' domain to write a correct and comprehensive set of spec for a task catalog. Specs also cannot detect if there are several new resources or properties that have not been described in the spec file.

Data-driven tests can offer an alternative way to ensure that there are no unwanted changes in the tasks catalogs. The idea behind them is building catalogs in human-readable format before and after the changes are made. Then these files can be compared and everything that have been changes will become visible.

Using the **-V** options will save the current catalog to the *catalogs* folder. These generated catalogs can be useful when reviewing complex patches with major changes to the modules or manifests. A diff for data changes may help a developer/reviewer examine the catalog contents and check that every resource or class are receiving the correct property values.

You can also use **-v** option to enable automatic catalog checks. It should be done after you have generated the initial versions and made some changes. Running the tests with this option enabled will generate the catalogs again and compare them to the saved version. If there are differences the test will be failed and you will be able to locate the failed tasks. Here is an example workflow one may use to examine a complex patch for data layer changes (we assume she is a cool ruby developer and use the rvm manager and bundler):

```
$ git clone https://github.com/openstack/fuel-library
$ cd fuel-library
$ rvm use ruby-2.1.3
$ PUPPET_GEM_VERSION=3.4.0
$ PUPPET_VERSION=3.4.0
$ ./tests/noop/setup_and_diagnostics.sh -B
$ ./deployment/remove_modules.sh && ./deployment/update_modules.sh
$ ./tests/noop/noop_tests.sh -V -j10 -b -s swift
$ git review -d $swift_die_hard_patch_id
$ ./tests/noop/noop_tests.sh -v -j10 -b -s swift
```

At this point, the reviewer will get the data diffs proposed to the swift modules and finish her thorough review. Note that the command *./tests/noop/setup_and_diagnostics.sh -B* gets a clean state and sets things up, while removing and updating_modules is required to make things go smooth.

Using external environment variables and custom paths

There are a number of environment variables used by either the task manager or by the specs themselves which can alter their behaviour and override the default or calculated values.

Paths related:

- **SPEC_ROOT_DIR** Set the path to the root folder of the framework. Many other folders are found relative to this path.
- **SPEC_SPEC_DIR** The path to the folder with the spec files. You can change it but it should be at the *spec/hosts* from the root folder or the rpsec-puppet will break.

- **SPEC_MODULE_PATH** or **SPEC_MODULEPATH** Set the path to the modules library. It can be either a path to a single directory with Puppet modules or a string with colon-separated paths to several module directories.
- **SPEC_TASK_DIR** Set the path to the task manifests folder.
- **SPEC_DEPLOYMENT_DIR** Set the path to the *deployment* directory. It's actually use only to find the scripts to update and reset modules.
- **SPEC_TASK_ROOT_DIR** Set the root path of the RSpec execution. RSpec command will be run from this directory. Usually it's the same dir as the **SPEC_ROOT_DIR**.
- **WORKSPACE** This variable is passed by the Jenkins jobs or will default to the *workspece* folder. Currently used only to store the Ruby gems installed by the *bundler* if *RVM* is not used.
- **SPEC_FACTS_DIR** The path to the folder with facts yaml files.
- **SPEC_HIERA_DIR** or **SPEC_YAML_DIR** The path to the folder with Hiera yaml files.

Spec related:

- **SPEC_FACTS_NAME** Set the name of the facts file that will be used by the spec process. It's set when the task is being run.
- **SPEC_HIERA_NAME** or **SPEC_ASTUTE_FILE_NAME** Set the name of the Hiera yaml file that will be used by the spec process. It's set when the task is being run.
- **SPEC_FILE_NAME** Set the spec/manifest file name for the spec process to test. It's set when the task is being run and even can override the internal value.
- **SPEC_BUNDLE_EXEC** Use *bundle exec* to run the *rspec* command by the task object.
- **SPEC_UPDATE_GLOBALS** Save the generated globals files instead of just checking that globals task's catalog is compiling without and error.
- **SPEC_CATALOG_SHOW** Ask the spec to output the catalog contents.
- **SPEC_SHOW_STATUS** Ask the spec to output the status text.

Debug related:

- **SPEC_TASK_CONSOLE** Run the pry console in the manager process.
- **SPEC_RSPEC_CONSOLE** Run the pry console in the RSpec process.
- **SPEC_PUPPET_DEBUG** Enable debug output of the Puppet's catalog compilation. This variable is also used by many other rspec suites of the Mirantis Puppet modules outside of the Noop tests framework to output the additional debug information.
- **SPEC_TASK_DEBUG** Enable the debug output of the task and manager objects.
- **SPEC_DEBUG_LOG** This variable can the the debug log destination file.

Fixtures source related:

- **NOOP_FIXTURES_REPO_URL** Fixtures repository. Defaults to *https://github.com/openstack/fuel-noop-fixtures.git*
- **NOOP_FIXTURES_BRANCH** Fixtures branch. Defaults to *origin/master*
- **NOOP_FIXTURES_GERRIT_URL** Gerrit repository. Defaults to *https://review.openstack.org/openstack/fuel-noop-fixtures*
- **NOOP_FIXTURES_GERRIT_COMMIT** Gerrit commit ref that should be cherry-picked. Could contain multiple refs, space separated. Defaults to *none*

Many of this variables can be set by the Noop manager CLI options, or you can always export them externally.

Using additional RSpec matchers and task helpers

There are some matchers for RSpec one would like to use

ensure_transitive_dependency(before, after)

This matcher allows one to check whether there is a dependency between *after* and *before* resources even if this dependency is transitional by means of several other resources or containers such as classes or defines.

CHAPTER 3

Search in this guide

- search