Validations Framework Client Documentation

Release 1.9.1.dev6

OpenStack LLC

Sep 12, 2023

CONTENTS

This is the Validations Framework Client API. It provides:

- a Python API: the `validations_libs` module, to
- list and run validation(s) on node(s).

CHAPTER

ONE

CONTENTS

1.1 validations-libs

A collection of python libraries for the Validation Framework

The validations will help detect issues early in the deployment process and prevent field engineers from wasting time on misconfiguration or hardware issues in their environments.

- Free software: Apache_license
- Documentation: https://docs.openstack.org/validations-libs/latest/
- Source: https://opendev.org/openstack/validations-libs
- Bugs - Upstream: https://bugs.launchpad.net/tripleo/+bugs?field.tag=validations
- Bugs - Downstream: https://bugzilla.redhat.com/buglist.cgi?component=validations-libs&product=Red%20Hat%20OpenStack

1.1.1 Development Environment Setup

Vagrantfiles for CentOS and Ubuntu have been provided for convenience; simply copy one into your desired location and rename to `Vagrantfile`, then run:

```
vagrant up
```

Once complete you will have a clean development environment ready to go for working with Validation Framework.

1.1.2 podman Quickstart

A Dockerfile is provided at the root of the Validations Library project in order to quickly set and hack the Validation Framework, on a equivalent of a single machine. Build the container from the Dockerfile by running:

```
podman build -t "vf:dockerfile" .
```

From the validations-libs repo directory.

Note: More complex images are available in the dockerfiles directory and require explicit specification of both build context and the Dockerfile.

Since the podman build uses code sourced from the buildah project to build container images. It is also possible to build an image using:

```
buildah bud -t "vf:dockerfile" .
```

Then you can run the container and start to run some builtin Validations:

```
podman run -ti vf:dockerfile /bin/bash
```

Then run validations:

```
validation.py run --validation check-ftype,512e --inventory /etc/ansible/hosts
```

1.1.3 Skip list

You can provide a file with a list of Validations to skip via the run command:

```
validation.py run --validation check-ftype,512e --inventory /etc/ansible/
↪hosts --skiplist my-skip-list.yaml
```

This file should be formed as:

```
validation-name:
  hosts: targeted_hostname
  reason: reason to ignore the file
  lp: bug number
```

The framework will skip the validation against the `hosts` key. In order to skip the validation on every hosts, you can set `all` value such as:

```
hosts: all
```

If no hosts key is provided for a given validation, it will be considered as `hosts: all`.

Note: The `reason` and `lp` key are for tracking and documentation purposes, the framework wont use those keys.

1.1.4 Community Validations

Community Validations enable a sysadmin to create and execute validations unique to their environment through the `validation` CLI.

The Community Validations will be created and stored in an unique, standardized and known place, called `'community-validations/'`, in the home directory of the non-root user which is running the CLI.

Note: The Community Validations are enabled by default. If you want to disable them, please set `[DEFAULT].enable_community_validations` to `False` in the validation configuration file located by default in `/etc/validation.cfg`

The first level of the mandatory structure will be the following (assuming the operator uses the `pennywise` user):

```
/home/pennywise/community-validations
 library
 lookup_plugins
 playbooks
 roles
```

Note: The `community-validations` directory and its sub directories will be created at the first CLI use and will be checked everytime a new community validation will be created through the CLI.

How To Create A New Community Validation

```
[pennywise@localhost]$ validation init my-new-validation
Validation config file found: /etc/validation.cfg
New role created successfully in /home/pennywise/community-validations/roles/
↪my_new_validation
New playbook created successfully in /home/pennywise/community-validations/
↪playbooks/my-new-validation.yaml
```

The `community-validations/` directory should have been created in the home directory of the `pennywise` user.

```
[pennywise@localhost ~]$ cd && tree community-validations/
community-validations/
 library
 lookup_plugins
 playbooks
ăă  my-new-validation.yaml
 roles
     my_new_validation
         defaults
        ăă  main.yml
         files
```

(continues on next page)

(continued from previous page)

```
        handlers
       ăă  main.yml
        meta
       ăă  main.yml
        README.md
        tasks
       ăă  main.yml
        templates
        tests
       ăă  inventory
       ăă  test.yml
        vars
            main.yml

13 directories, 9 files
```

Your new community validation should also be available when listing all the validations available on your system.

```
[pennywise@localhost ~]$ validation list
Validation config file found: /etc/validation.cfg
+------------------------------+----------------------------------+------------
↪--------------------+-----------------------------------+---------------+
| ID                           | Name                             | Groups     ␣
↪                    | Categories                        | Products      |
+------------------------------+----------------------------------+------------
↪--------------------+-----------------------------------+---------------+
| 512e                         | Advanced Format 512e Support     | ['prep',
↪'pre-deployment']     | ['storage', 'disk', 'system']     | ['common']    |
| check-cpu                    | Verify if the server fits the    | ['prep',
↪'backup-and-restore', | ['system', 'cpu', 'core', 'os']   | ['common']    |
|                              | CPU core requirements            | 'pre-
↪introspection']           |                                   |               ␣
↪  |
| check-disk-space-pre-upgrade  | Verify server fits the disk     | ['pre-
↪upgrade']               | ['system', 'disk', 'upgrade']     | ['common']  ␣
↪ |
|                              | space requirements to perform    |               ␣
↪                    |                                   |               |
|                              | an upgrade                       |               ␣
↪                    |                                   |               |
| check-disk-space             | Verify server fits the disk      | ['prep',
↪'pre-introspection']  | ['system', 'disk', 'upgrade']     | ['common']    |
|                              | space requirements               |               ␣
↪                    |                                   |               |
| check-ftype                  | XFS ftype check                  | ['pre-
↪upgrade']               | ['storage', 'xfs', 'disk']        | ['common']  ␣
↪ |
| check-latest-packages-version | Check if latest version of      | ['pre-
↪upgrade']               | ['packages', 'rpm', 'upgrade']    | ['common']  ␣
```

(continues on next page)

(continued from previous page)

```
↪ |
|                              | packages is installed          |          ␣
↪                    |                                    |              |
| check-ram                    | Verify the server fits the RAM | ['prep',
↪'pre-introspection',  | ['system', 'ram', 'memory', 'os'] | ['common']    |
|                              | requirements                   | 'pre-
↪upgrade']                  |                                |          ␣
↪  |
| check-selinux-mode           | SELinux Enforcing Mode Check   | ['prep',
↪'pre-introspection']  | ['security', 'selinux']           | ['common']    |
| dns                          | Verify DNS                     | ['pre-
↪deployment']             | ['networking', 'dns']             | ['common']  ␣
↪ |
| no-op                        | NO-OP validation               | ['no-op']  ␣
↪                   | ['noop', 'dummy', 'test']         | ['common']    |
| ntp                          | Verify all deployed servers    | ['post-
↪deployment']           | ['networking', 'time', 'os']      | ['common']   ␣
↪|
|                              | have their clock synchronised  |          ␣
↪                    |                                    |              |
| service-status               | Ensure services state          | ['prep',
↪'backup-and-restore', | ['systemd', 'container',          | ['common']    |
|                              |                                | 'pre-
↪deployment', 'pre-        | 'docker', 'podman']               |          ␣
↪  |
|                              |                                | upgrade',
↪'post-deployment',   |                                  |              |
|                              |                                | 'post-
↪upgrade']               |                                  |          ␣
↪ |
| validate-selinux             | validate-selinux               | ['backup-
↪and-restore', 'pre-   | ['security', 'selinux', 'audit']  | ['common']    |
|                              |                                | deployment
↪', 'post-            |                                  |              |
|                              |                                | deployment
↪', 'pre-upgrade',    |                                  |              |
|                              |                                | 'post-
↪upgrade']               |                                  |          ␣
↪ |
| my-new-validation            | Brief and general description  | ['prep',
↪'pre-deployment']     | ['networking', 'security', 'os',  | ['community'] |
|                              | of the validation              |          ␣
↪                    | 'system']                         |              |
+------------------------------+--------------------------------+------------
↪--------------------+-----------------------------------+---------------+
```

To get only the list of your community validations, you can filter by products:

```
[pennywise@localhost]$ validation list --product community
Validation config file found: /etc/validation.cfg
```

(continues on next page)

(continued from previous page)

```
+------------------+-----------------------------------------+-------------
↪-------------+-----------------------------------------+---------------+
| ID               | Name                                    | Groups      ␣
↪             | Categories                              | Products      |
+------------------+-----------------------------------------+-------------
↪-------------+-----------------------------------------+---------------+
| my-new-validation | Brief and general description of the    | ['prep',
↪'pre-deployment'] | ['networking', 'security', 'os',        | ['community
↪'] |
|                  | validation                              |             ␣
↪             | 'system']                               |               |
+------------------+-----------------------------------------+-------------
↪-------------+-----------------------------------------+---------------+
```

How To Develop Your New Community Validation

As you can see above, the `validation init` CLI sub command has generated a new Ansible role by using ansible-galaxy and a new Ansible playbook in the `community-validations/` directory.

> **Warning:** The community validations wont be supported at all. We wont be responsible as well for potential use of malignant code in their validations. Only the creation of a community validation structure through the new Validation CLI sub command will be supported.

You are now able to implement your own validation by editing the generated playbook and adding your ansible tasks in the associated role.

For people not familiar with how to write a validation, get started with this documentation.

1.2 Contributing to validations-libs

Contributions to validations-libs follow guidelines largely similar to those of other openstack projects.

If youre interested in contributing to the validations-libs project, the following will help get you started:

> https://docs.openstack.org/infra/manual/developers.html

If you already have a good understanding of how the system works and your OpenStack accounts are set up, you can skip to the development workflow section of this documentation to learn how changes to OpenStack should be submitted for review via the Gerrit tool:

> https://docs.openstack.org/infra/manual/developers.html#development-workflow

Pull requests submitted through GitHub will be ignored.

Validations are meant to verify functionality of tripleo systems. Therefore a special care should be given to testing your code before submitting a review.

1.2.1 Branches and version management

Validation Framework project uses semantic versioning and derives names of stable branches from the released minor versions. The latest minor version released is the only exception as it is derived from the *master* branch.

Therefore, all code used by version 1.n.* of the project resides in *stable/1.n* branch, and when version 1.(n+1) is released, new branch *stable/1.(n+1)* will be created.

By default, stable branches recieve only bug fixes and feature backports are decided on case basis after all the necessary discussions and procedures have taken place.

Communication

- IRC channel `#validation-framework` at Libera (For all subject-matters)
- IRC channel `#tripleo` at OFTC (OpenStack and TripleO discussions)

Contributor License Agreement

In order to contribute to the validations-libs project, you need to have signed OpenStacks contributors agreement.

See also:

- https://docs.openstack.org/infra/manual/developers.html
- https://wiki.openstack.org/wiki/CLA

Project Hosting Details

Code Hosting
https://opendev.org/openstack/validations-libs

Code Review
https://review.opendev.org/#/q/status:open+project:openstack/validations-libs,n,z

1.3 Testing

1.3.1 Python Guideline Enforcement

All code has to pass the pep8 style guideline to merge into OpenStack, to validate the code against these guidelines you can run:

```
$ tox -e pep8
```

1.3.2 Unit Testing

It is strongly encouraged to run the unit tests locally under one or more test environments prior to submitting a patch. To run all the recommended environments sequentially and pep8 style guideline run:

```
$ tox
```

You can also selectively pick specific test environments by listing your chosen environments after a -e flag:

```
$ tox -e py36,py38,pep8
```

Note: Tox sets up virtual environment and installs all necessary dependencies. Sharing the environment with devstack testing is not recommended due to conflicting configuration with system dependencies.

1.4 Validations Framework Command Line Interface (CLI)

1.4.1 Global Options

Validations Framework Command Line Interface (CLI)

```
validation [--version] [-v | -q] [--log-file LOG_FILE] [--debug]
```

--version
: show programs version number and exit

-v, --verbose
: Increase verbosity of output. Can be repeated.

-q, --quiet
: Suppress output except warnings and errors.

--log-file <LOG_FILE>
: Specify a file to log output. Disabled by default.

--debug
: Show tracebacks on errors.

1.4.2 Command Options

init

Initialize Community Validation Skeleton

```
validation init
    [--config CONFIG]
    [--validation-dir VALIDATION_DIR]
    [--ansible-base-dir ANSIBLE_BASE_DIR]
    <validation_name>
```

--config <CONFIG>

Config file path for Validation Framework.

--validation-dir <VALIDATION_DIR>

Path where the validation playbooks is located.

--ansible-base-dir <ANSIBLE_BASE_DIR>

Path where the ansible roles, library and plugins are located.

validation_name

The name of the Community Validation: Validation name is limited to contain only lowercase alphanumeric characters, plus _ or - and starts with an alpha character. Ex: my-val, my_val2. This will generate an Ansible role and a playbook in /home/zuul/src/opendev.org/openstack/validations-libs/.tox/docs/community-validations. Note that the structure of this directory will be created at the first use.

This command is provided by the validations-libs plugin.

1.5 Full Validations-libs Python API Reference

1.5.1 validations_libs

validations_libs package

Subpackages

validations_libs.callback_plugins package

Submodules

validations_libs.callback_plugins.vf_fail_if_no_hosts module

class validations_libs.callback_plugins.vf_fail_if_no_hosts.**CallbackModule**(**args: Any*, ***kwargs: Any*)

Bases: CallbackBase

CALLBACK_NAME = 'fail_if_no_hosts'

CALLBACK_VERSION = 2.0

v2_playbook_on_stats(*stats*)

validations_libs.callback_plugins.vf_http_json module

class validations_libs.callback_plugins.vf_http_json.**CallbackModule**(**args: Any*, ***kwargs: Any*)

Bases: *CallbackModule*

CALLBACK_NAME = 'http_json'

CALLBACK_NEEDS_WHITELIST = True

CALLBACK_TYPE = 'aggregate'

CALLBACK_VERSION = 2.0

v2_playbook_on_stats(*stats*)

Display info about playbook statistics

validations_libs.callback_plugins.vf_http_json.**current_time**()

validations_libs.callback_plugins.vf_http_json.**http_post**(*data*)

validations_libs.callback_plugins.vf_validation_json module

class validations_libs.callback_plugins.vf_validation_json.**CallbackModule**(**args: Any*, ***kwargs: Any*)

Bases: CallbackBase

CALLBACK_NAME = 'validation_json'

CALLBACK_NEEDS_WHITELIST = True

CALLBACK_TYPE = 'aggregate'

CALLBACK_VERSION = 2.0

v2_playbook_on_handler_task_start(*task*)

v2_playbook_on_no_hosts_matched()

v2_playbook_on_play_start(*play*)

v2_playbook_on_start(*playbook*)

v2_playbook_on_stats(*stats*)

Display info about playbook statistics

v2_playbook_on_task_start(*task*, *is_conditional*)

validations_libs.callback_plugins.vf_validation_json.**current_time**()

validations_libs.callback_plugins.vf_validation_json.**secondsToStr**(*t*)

validations_libs.callback_plugins.vf_validation_output module

class validations_libs.callback_plugins.vf_validation_output.**CallbackModule**(**args: Any*, ***kwargs: Any*)

Bases: CallbackBase

CALLBACK_NAME = 'validation_output'

CALLBACK_TYPE = 'stdout'

CALLBACK_VERSION = 2.0

print_failure_message(*host_name*, *task_name*, *results*, *abridged_result*)

Print a human-readable error info from Ansible result dictionary.

v2_playbook_on_play_start(*play*)

v2_playbook_on_stats(*stats*)

v2_playbook_on_task_start(*task*, *is_conditional*)

v2_runner_on_failed(*result*, ***kwargs*)

v2_runner_on_ok(*result*, ***kwargs*)

v2_runner_on_skipped(*result*, ***kwargs*)

v2_runner_on_unreachable(*result*, ***kwargs*)

validations_libs.callback_plugins.vf_validation_output.**indent**(*text*)

Indent the given text by four spaces.

validations_libs.callback_plugins.vf_validation_stdout module

class validations_libs.callback_plugins.vf_validation_stdout.**CallbackModule**(**args: Any*, ***kwargs: Any*)

Bases: CallbackBase

CALLBACK_NAME = 'validation_stdout'

CALLBACK_TYPE = 'stdout'

CALLBACK_VERSION = 2.0

validations_libs.callback_plugins.vf_validation_stdout.**current_time**()

validations_libs.callback_plugins.vf_validation_stdout.**secondsToStr**(*t*)

Module contents

This module contains various callbacks developed to facilitate functions of the Validation Framework.

Somewhat unorthodox naming of the callback classes is a direct result of how ansible handles loading plugins. The ansible determines the purpose of each plugin by looking at its class name. As you can see in the https://github.com/ansible/ansible/blob/devel/lib/ansible/plugins/loader.py from the ansible repo, the loader uses the class names to categorize plugins. This means that every callback plugin has to have the same class name, and the unfortunate coder has to discern their purpose by checking their module names.

validations_libs.cli package

Submodules

validations_libs.cli.app module

class validations_libs.cli.app.**ValidationCliApp**

Bases: App

Cliff application for the *ValidationCli* tool. :param description: one-liner explaining the program purpose :param version: application version number :param command_manager: plugin loader :param deferred_help: Allow subcommands to accept *help* with allowing to defer help print after initialize_app

clean_up(*cmd*, *result*, *err*)

Hook run after a command is done to shutdown the app.

Parameters

- **cmd** (*cliff.command.Command*) command processor being invoked
- **result** (*int*) return value of cmd
- **err** (*Exception*) exception or None

initialize_app(*argv*)

Hook for subclasses to take global initialization action after the arguments are parsed but before a command is run. Invoked only once, even in interactive mode.

Parameters

argv List of arguments, including the subcommand to run. Empty for interactive mode.

prepare_to_run_command(*cmd*)

Perform any preliminary work needed to run a command.

Parameters

cmd (*cliff.command.Command*) command processor being invoked

validations_libs.cli.app.**main**(*argv=['-b', 'latex', 'doc/source', 'doc/build/pdf']*)

validations_libs.cli.base module

class validations_libs.cli.base.**Base**

Bases: object

Base class for CLI arguments management

config = {}

config_section = ['default', 'ansible_runner', 'ansible_environment']

set_argument_parser(*vf_parser*, *args*)

Set Arguments parser depending of the precedence ordering: * User CLI arguments * Configuration file * Default CLI values

class validations_libs.cli.base.**BaseCommand**(*app*, *app_args*, *cmd_name=None*)

Bases: Command

Base Command client implementation class

get_parser(*prog_name*)

Argument parser for base command

class validations_libs.cli.base.**BaseLister**(*app*, *app_args*, *cmd_name=None*)

Bases: Lister

Base Lister client implementation class

get_parser(*prog_name*)

Argument parser for base lister

class validations_libs.cli.base.**BaseShow**(*app*, *app_args*, *cmd_name=None*)

Bases: ShowOne

Base Show client implementation class

get_parser(*parser*)

Argument parser for base show

validations_libs.cli.colors module

validations_libs.cli.colors.**color_output**(*output*, *status=None*)

Apply color to output based on colors dict entries. Unknown status or no status at all results in aplication of YELLOW color.

Note: Coloring itself is performed using format method of the string class. This function is merely a wrapper around it, and around ANSI escape sequences as defined by ECMA-48.

---

validations_libs.cli.common module

class validations_libs.cli.common.**Spinner**(*delay=None*)

Bases: object

Animated spinner to indicate activity during processing

busy = False

delay = 0.1

spinner_task()

static spinning_cursor()

class validations_libs.cli.common.**ValidationHelpFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: ArgumentDefaultsHelpFormatter, SmartHelpFormatter

Composite CLI help formatter, providing both default argument values, and correct new line treatment.

validations_libs.cli.common.**print_dict**(*data*)

Print table from python dict with PrettyTable

validations_libs.cli.common.**read_cli_data_file**(*data_file*)

Read CLI data (YAML/JSON) file. :param data_file: Path to the requested file. :type data_file: path like

Returns
Parsed YAML/JSON file

Return type
dict

Raises
RuntimeError if the file doesnt exist or is malformed.

validations_libs.cli.common.**write_junitxml**(*output_junitxml*, *results*)

Write output file as JUnitXML format

validations_libs.cli.common.**write_output**(*output_log*, *results*)

Write output log file as Json format

validations_libs.cli.community module

class validations_libs.cli.community.**CommunityValidationInit**(*app*, *app_args*, *cmd_name=None*)

Bases: *BaseCommand*

Initialize Community Validation Skeleton

get_parser(*parser*)

Argument parser for Community Validation Init

take_action(*parsed_args*)

Take Community Validation Action

validations_libs.cli.constants module

Constants for the VF CLI. Constains larger, more frequently used and redundant CLI help strings.

validations_libs.cli.file module

class validations_libs.cli.file.**File**(*app*, *app_args*, *cmd_name=None*)

Bases: *BaseCommand*

Include and exclude validations by name(s), group(s), category(ies) or by product(s) and run them from File

get_parser(*parser*)

Argument parser for validation file

take_action(*parsed_args*)

Take action

validations_libs.cli.history module

class validations_libs.cli.history.**GetHistory**(*app*, *app_args*, *cmd_name=None*)

Bases: *BaseCommand*

Display details about a specific Validation execution

get_parser(*parser*)

Argument parser for base command

take_action(*parsed_args*)

Override to do something useful.

The returned value will be returned by the program.

class validations_libs.cli.history.**ListHistory**(*app*, *app_args*, *cmd_name=None*)

Bases: *BaseLister*

Display Validations execution history

get_parser(*parser*)

Argument parser for base lister

take_action(*parsed_args*)

Run command.

Return a tuple containing the column names and an iterable containing the data to be listed.

validations_libs.cli.lister module

class validations_libs.cli.lister.**ValidationList**(*app*, *app_args*, *cmd_name=None*)

Bases: *BaseLister*

List the Validations Catalog

get_parser(*parser*)

Argument parser for validation run

take_action(*parsed_args*)

Take validation action

validations_libs.cli.parseractions module

class validations_libs.cli.parseractions.**CommaListAction**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Bases: Action

class validations_libs.cli.parseractions.**KeyValueAction**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Bases: Action

A custom action to parse arguments as key=value pairs Ensures that dest is a dict and values are strings.

validations_libs.cli.run module

class validations_libs.cli.run.**Run**(*app*, *app_args*, *cmd_name=None*)

Bases: *BaseCommand*

Run Validations by name(s), group(s), category(ies) or by product(s)

get_parser(*parser*)

Argument parser for validation run

take_action(*parsed_args*)

Take validation action

validations_libs.cli.show module

class validations_libs.cli.show.**Show**(*app*, *app_args*, *cmd_name=None*)

Bases: *BaseShow*

Show detailed informations about a Validation

get_parser(*parser*)

Argument parser for validation show

take_action(*parsed_args*)

Take validation action

class validations_libs.cli.show.**ShowGroup**(*app*, *app_args*, *cmd_name=None*)

Bases: *BaseLister*

Show detailed informations about Validation Groups

get_parser(*parser*)

Argument parser for validation show group

take_action(*parsed_args*)

Take validation action

class validations_libs.cli.show.**ShowParameter**(*app*, *app_args*, *cmd_name=None*)

Bases: *BaseShow*

Show Validation(s) parameter(s)

Display Validation(s) Parameter(s) which could be overriden during an execution. It could be filtered by **validation_id**, **group(s)**, **category(ies)** or by **products**.

get_parser(*parser*)

Argument parser for base show

take_action(*parsed_args*)

Return a two-part tuple with a tuple of column names and a tuple of values.

Module contents

validations_libs.community package

Submodules

validations_libs.community.init_validation module

class validations_libs.community.init_validation.**CommunityValidation**(*validation_name*, *validation_dir='/usr/share/ar playbooks'*, *ansible_base_dir='/usr/share/ans*

Bases: `object`

Init Community Validation Role and Playbook Command Class

Initialize a new community role using ansible-galaxy and create a playboook from a template.

create_playbook(*content="---\n# This playbook has been generated by the 'validation init' CLI.\n#\n# As shown here in this template, the validation playbook requires three\n# top-level directive:\n# "hosts", "vars -> metadata" and "roles".\n#\n# "hosts": specifies which nodes to run the validation on. The options can\n# be "all" (run on all nodes), or you could use the hosts defined\n# in the inventory.\n# "vars": this section serves for storing variables that are going to be\n# available to the Ansible playbook. The validations API uses the\n# "metadata" section to read each validation's name and description\n# These values are then reported by the API.\n#\n# The validations can be grouped together by specyfying a "groups" metadata.\n# Groups function similar to tags and a validation can thus be part of many\n# groups. To get a full list of the groups available and their description,\n# please run the following command on your Ansible Controller host:\n#\n# $ validation show group\n#\n# The validations can also be categorized by technical domain and acan belong to\n# one or multiple "categories". For example, if your validation checks some\n# networking related configuration, you may want to put "networking" as a\n# category. Note that this section is open and you are free to categorize your\n# validations as you like.\n#\n# The "products" section refers to the product on which you would like to run\n# the validation. It's another way to categorized your community validations.\n# Note that, by default, "community" is set in the "products" section to\n# help you list your validations by filtering by products:\n#\n# $ validation list --product community\n#\n- hosts: hostname\n gather_facts: false\n vars:\n metadata:\n name: Brief and general description of the validation\n description: |\n The complete description of this validation should be here\n# GROUPS:\n# Run "validation show group" to get the list of groups\n# :type group: 'list'\n# If you don't want to add groups for your validation, just\n# set an empty list to the groups key\n groups: []\n# CATEGORIES:\n# :type group: 'list'\n# If you don't want to categorize your validation, just\n# set an empty list to the categories key\n categories: []\n products:\n - community\n roles:\n - {}\n"*)

Create the playbook for the new community validation

execute()

Execute the actions necessary to create a new community validation

Check if the role name is compliant with Ansible specification Initializing the new role using ansible-galaxy Creating the validation playbook from a template on disk

Return type
NoneType

is_community_validations_enabled(*base_config*)

Checks if the community validations are enabled in the config file

Parameters
base_config (Dict) Contents of the configuration file

Return type
Boolean

`is_playbook_exists()`

New playbook existence check

This class method checks if the new playbook file is already existing in the official validations catalog and in the current community validations directory.

First, it gets the list of the playbooks yaml file available in `constants.ANSIBLE_VALIDATIONS_DIR`. If there is a match in at least one of the directories, it returns `True`, otherwise `False`.

Return type
`Boolean`

`is_role_exists()`

New role existence check

This class method checks if the new role name is already existing in the official validations catalog and in the current community validations directory.

First, it gets the list of the role names available in `constants.ANSIBLE_ROLES_DIR`. If there is a match in at least one of the directories, it returns `True`, otherwise `False`.

Return type
`Boolean`

`property is_role_name_compliant`

Check if the role name is compliant with Ansible Rules

Roles Name are limited to contain only lowercase alphanumeric characters, plus _ and start with an alpha character.

Return type
`Boolean`

`property playbook_basedir`

Returns the absolute path of the community playbooks directory

Return type
`pathlib.PosixPath`

`property playbook_name`

Return the new playbook name with the yaml extension

Return type
`str`

`property playbook_path`

Returns the absolute path of the new community playbook yaml file

Return type
`pathlib.PosixPath`

`property role_basedir`

Returns the absolute path of the community validations roles

Return type
`pathlib.PosixPath`

property role_dir_path

Returns the community validation role directory name

Return type
pathlib.PosixPath

property role_name

Returns the community validation role name

Return type
str

Module contents

Submodules

validations_libs.ansible module

class validations_libs.ansible.**Ansible**(*uuid=None*)

Bases: object

An Object for encapsulating an Ansible execution

run(*playbook*, *inventory*, *workdir*, *playbook_dir=None*, *connection='smart'*, *output_callback=None*, *base_dir='/usr/share/ansible'*, *ssh_user=None*, *key=None*, *module_path=None*, *limit_hosts=None*, *tags=None*, *skip_tags=None*, *verbosity=0*, *quiet=False*, *extra_vars=None*, *gathering_policy='smart'*, *extra_env_variables=None*, *parallel_run=False*, *callback_whitelist=None*, *ansible_cfg_file=None*, *ansible_timeout=30*, *ansible_artifact_path=None*, *log_path=None*, *run_async=False*, *python_interpreter=None*, *validation_cfg_file=None*)

Execute one or multiple Ansible playbooks

Parameters

- **playbook** (string) The Absolute path of the Ansible playbook
- **inventory** (string) Either proper inventory file or a comma-separated list
- **workdir** (string) The absolute path of the Ansible-runner artifacts directory
- **playbook_dir** (string) The absolute path of the Validations playbooks directory
- **connection** (*String*) Connection type (local, smart, etc). (efaults to smart)
- **output_callback** (string) Callback for output format. Defaults to yaml.
- **base_dir** (string) The absolute path of the default validations base directory
- **ssh_user** (string) User for the ssh connection (Defaults to root)
- **key** (string) Private key to use for the ssh connection.
- **module_path** (string) Location of the ansible module and library.

- **limit_hosts** (string) Limit the execution to the hosts.
- **tags** (string) Run specific tags.
- **skip_tags** (string) Skip specific tags.
- **verbosity** (integer) Verbosity level for Ansible execution.
- **quiet** (boolean) Disable all output (Defaults to False)
- **extra_vars** (*Either a Dict or the absolute path of JSON or YAML*) Set additional variables as a Dict or the absolute path of a JSON or YAML file type.
- **gathering_policy** This setting controls the default policy of fact gathering (smart, implicit, explicit). (Defaults to smart)
- **extra_env_vars** (dict) Set additional ansible variables using an extravar dictionary.
- **parallel_run** (boolean) Isolate playbook execution when playbooks are to be executed with multi-processing.
- **callback_whitelist** (list or string) Comma separated list of callback plugins. Custom output_callback is also whitelisted. (Defaults to None)
- **ansible_cfg_file** (string) Path to an ansible configuration file. One will be generated in the artifact path if this option is None.
- **ansible_timeout** (integer) Timeout for ansible connections. (Defaults to 30 minutes)
- **ansible_artifact_path** (string) The Ansible artifact path
- **log_path** (string) The absolute path of the validations logs directory
- **run_async** (boolean) Enable the Ansible asynchronous mode (Defaults to False)
- **python_interpreter** (string) Path to the Python interpreter to be used for module execution on remote targets, or an automatic discovery mode (auto, auto_silent or the default one auto_legacy)
- **validation_cfg_file** (dict) A dictionary of configuration for Validation loaded from an validation.cfg file.

Returns

A tuple containing the the absolute path of the executed playbook, the return code and the status of the run

Return type

tuple

validations_libs.constants module

Default paths for validation playbook directory, validation groups definitions and validation logs are defined here.

These paths are used in an absence of user defined overrides, or as a fallback, when custom locations fail.

validations_libs.exceptions module

This module contains Validation Framework specific exceptions, to be raised by Validation Framework runtime.

The exceptions are meant to cover the most common of the possible fail states the framework can encounter, with the rest evoking one of the built in exceptions, such as RuntimeError. Use of these exceptions should be limited to cases when cause is known and within the context of the framework itself.

exception validations_libs.exceptions.**ValidationRunException**

Bases: Exception

ValidationRunException is to be raised when actions initiated by the CLI run subcommand or *run_validations* method of the *ValidationsActions* class, cause unacceptable behavior from which it is impossible to recover.

exception validations_libs.exceptions.**ValidationShowException**

Bases: Exception

ValidationShowException is to be raised when actions initiated by the CLI show subcommands or *show_history*, *show_validations* or *show_validations_parameters* methods of the *ValidationsActions* class, cause unacceptable behavior from which it is impossible to recover.

validations_libs.group module

class validations_libs.group.**Group**(*groups*)

Bases: object

An object for encapsulating the groups of validation

The validations can be grouped together by specifying a `groups` metadata. These `groups` are referenced in a `groups.yaml` file on the filesystem.

```
group1:
- description: >-
    Description of the group1
group2:
- description: >-
    Description of the group2
group3:
- description: >-
    Description of the group3
```

property get_data

Get the full content of the `groups.yaml` file

Returns
The content of the `groups.yaml` file

Return type
dict

Example

```
>>> groups = "/foo/bar/groups.yaml"
>>> grp = Group(groups)
>>> print(grp.get_data)
{'group1': [{'description': 'Description of the group1'}],
 'group2': [{'description': 'Description of the group2'}],
 'group3': [{'description': 'Description of the group3'}]}
```

property get_formated_groups

Get a formated list of groups for output display

Returns
information about parsed groups

Return type
list of *tuples*

Example

```
>>> groups = "/foo/bar/groups.yaml"
>>> grp = Group(groups)
>>> print(grp.get_formated_group)
[('group1', 'Description of the group1'),
 ('group2', 'Description of the group2'),
 ('group3', 'Description of the group3')]
```

property get_groups_keys_list

Get the list of the group name only

Returns
The list of the group name

Return type
list

Example

```
>>> groups = "/foo/bar/groups.yaml"
>>> grp = Group(groups)
>>> print(grp.get_groups_keys_list)
['group1', 'group2', 'group3']
```

validations_libs.logger module

validations_libs.logger.**getLogger**(*loggerName*, *stream_lvl=30*)

Create logger instance.

Parameters

- **loggerName** (*str*) name of the new Logger instance
- **stream_lvl** (*int*) minimum level at which the messages will be printed to stream

Return type
Logger

validations_libs.utils module

validations_libs.utils.**check_community_validations_dir**(*basedir=PosixPath('/home/zuul/src/opendev.org/openstack/validations-libs/.tox/docs/community-validations')*, *subdirs=[PosixPath('/home/zuul/src/opendev.org/openstack/validations-libs/.tox/docs/community-validations/roles'), PosixPath('/home/zuul/src/opendev.org/openstack/validations-libs/.tox/docs/community-validations/playbooks'), PosixPath('/home/zuul/src/opendev.org/openstack/validations-libs/.tox/docs/community-validations/library'), PosixPath('/home/zuul/src/opendev.org/openstack/validations-libs/.tox/docs/community-validations/lookup_plugins')]*)

Check presence of the community validations directory structure

The community validations are stored and located in:

```
/home/<username>/community-validations
 library
 lookup_plugins
 playbooks
 roles
```

This function checks for the presence of the community-validations directory in the $HOME of the user running the validation CLI. If the primary directory doesnt exist, this function will create it and will check if the four subdirectories are present and will create them otherwise.

Parameters

- **basedir** (`pathlib.PosixPath`) Absolute path of the community validations
- **subdirs** (`list` of `pathlib.PosixPath`) List of Absolute path of the community validations subdirs

Return type
NoneType

validations_libs.utils.**community_validations_on**(*validation_config*)

Check for flag for community validations to be enabled The default value is true

Parameters
validation_config (dict) A dictionary of configuration for Validation loaded from an validation.cfg file.

Returns
A boolean with the status of community validations flag

Return type
bool

validations_libs.utils.**create_artifacts_dir**(*log_path='/home/zuul/src/opendev.org/openstack/validations-libs/.tox/docs/validations'*, *prefix=''*)

Create Ansible artifacts directory for the validation run :param log_path: Directory asbolute path :type log_path: *string* :param prefix: Playbook name :type prefix: *string* :return: UUID of the validation run, absolute path of the validation artifacts directory :rtype: *string*, *string*

validations_libs.utils.**create_log_dir**(*log_path='/home/zuul/src/opendev.org/openstack/validations-libs/.tox/docs/validations'*)

Check for presence of the selected validations log dir. Create the directory if needed, and use fallback if that proves too tall an order.

Log the failure if encountering OSError or PermissionError.

Parameters
log_path (*string*) path of the selected log directory

Returns
valid path to the log directory

Return type
string

Raises
RuntimeError if even the fallback proves unavailable.

validations_libs.utils.**current_time**()

Return current time

validations_libs.utils.**find_config_file**(*config_file_name='validation.cfg'*)

Find the config file for Validation in the following order: * environment validation VALIDATION_CONFIG * current user directory * user home directory * Python prefix path which has been used for the installation * /etc/validation.cfg

validations_libs.utils.**get_validation_group_name_list**(*groups_path=None*)

Get the validation group name list only

Params groups_path
The path the groups.yaml file

Returns
The group name list

Return type
list

Example

```
>>> get_validation_group_name_list()
['group1',
 'group2',
 'group3',
 'group4']
```

validations_libs.utils.**get_validation_parameters**(*validation*)

Return dictionary of parameters

validations_libs.utils.**get_validations_data**(*validation*, *path='/usr/share/ansible/validation-playbooks'*, *validation_config=None*)

Return validation data with format:

ID, Name, Description, Groups, Parameters

Parameters

- **validation** (*string*) Name of the validation without the *yaml* extension. Defaults to *constants.ANSIBLE_VALIDATION_DIR*
- **path** (*string*) The path to the validations directory
- **validation_config** (dict) A dictionary of configuration for Validation loaded from an validation.cfg file.

Returns
The validation data with the format (ID, Name, Description, Groups, Parameters)

Return type
dict

Example

```
>>> validation = 'check-something'
>>> get_validations_data(validation)
{'Description': 'Verify that the server has enough something',
 'Groups': ['group1', 'group2'],
 'Categories': ['category1', 'category2'],
 'products': ['product1', 'product2'],
 'ID': 'check-something',
 'Name': 'Verify the server fits the something requirements',
 'Parameters': {'param1': 24}}
```

validations_libs.utils.**get_validations_parameters**(*validations_data*, *validation_name=None*, *groups=None*, *categories=None*, *products=None*)

Return parameters for a list of validations

Parameters

- **validations_data** (*list*) A list of absolute validations playbooks path
- **validation_name** (*list*) A list of validation name
- **groups** (*list*) A list of validation groups
- **categories** (*list*) A list of validation categories
- **products** (*list*) A list of validation products

Returns
a dictionary containing the current parameters for each *validation_name* or *groups*

Return type
dict

Example

```
>>> validations_data = ['/foo/bar/check-ram.yaml',
                        '/foo/bar/check-cpu.yaml']
>>> validation_name = ['check-ram', 'check-cpu']
>>> get_validations_parameters(validations_data, validation_name)
{'check-cpu': {'parameters': {'minimal_cpu_count': 8}},
 'check-ram': {'parameters': {'minimal_ram_gb': 24}}}
```

validations_libs.utils.**get_validations_playbook**(*path*, *validation_id=None*, *groups=None*, *categories=None*, *products=None*, *validation_config=None*)

Get a list of validations playbooks paths either by their names, their groups, by their categories or by their products.

Parameters

- **path** (*string*) Path of the validations playbooks
- **validation_id** (*list*) List of validation name
- **groups** (*list*) List of validation group
- **categories** (*list*) List of validation category
- **products** (*list*) List of validation product
- **validation_config** (dict) A dictionary of configuration for Validation loaded from an validation.cfg file.

Returns
A list of absolute validations playbooks path

Return type
list

Example

```
>>> path = '/usr/share/validation-playbooks'
>>> validation_id = ['512e','check-cpu']
>>> groups = None
>>> categories = None
```

(continues on next page)

(continued from previous page)

```
>>> products = None
>>> get_validations_playbook(path=path,
                             validation_id=validation_id,
                             groups=groups,
                             categories=categories,
                             products=products)
['/usr/share/ansible/validation-playbooks/512e.yaml',
 '/usr/share/ansible/validation-playbooks/check-cpu.yaml',]
```

validations_libs.utils.**load_config**(*config*)

Load Config File from CLI

validations_libs.utils.**parse_all_validations_on_disk**(*path*, *groups=None*, *categories=None*, *products=None*, *validation_config=None*)

Return a list of validations metadata which can be sorted by Groups, by Categories or by Products.

Parameters

- **path** (*string*) The absolute path of the validations directory
- **groups** (*list*) Groups of validations
- **categories** (*list*) Categories of validations
- **products** (*list*) Products of validations
- **validation_config** (dict) A dictionary of configuration for Validation loaded from an validation.cfg file.

Returns

A list of validations metadata.

Return type

list

Example

```
>>> path = '/foo/bar'
>>> parse_all_validations_on_disk(path)
[{'categories': ['storage'],
  'products': ['product1'],
  'description': 'Detect whether the node disks use Advanced Format.',
  'groups': ['prep', 'pre-deployment'],
  'id': '512e',
  'name': 'Advanced Format 512e Support'},
 {'categories': ['system'],
  'products': ['product1'],
  'description': 'Make sure that the server has enough CPU cores.',
  'groups': ['prep', 'pre-introspection'],
  'id': 'check-cpu',
  'name': 'Verify if the server fits the CPU core requirements'}]
```

validations_libs.utils.**read_validation_groups_file**(*groups_path=None*)

Load groups.yaml file and return a dictionary with its contents

Params groups_path
The path the groups.yaml file

Returns
The group list with their descriptions

Return type
dict

Example

```
>>> read_validation_groups_file()
{'group1': [{'description': 'Group1 description.'}],
 'group2': [{'description': 'Group2 description.'}]}
```

validations_libs.utils.**run_command_and_log**(*log*, *cmd*, *cwd=None*, *env=None*)

Run command and log output

Parameters

- **log** (*Logger*) Logger instance for logging
- **cmd** (`String`) Command to run in list form
- **cwd** Current working directory for execution
- **env** (`List`) Modified environment for command run

validations_libs.validation module

class validations_libs.validation.**Validation**(*validation_path*)

Bases: `object`

An object for encapsulating a validation

Each validation is an *Ansible* playbook. Each playbook have some `metadata`. Here is what a minimal validation would look like:

```
- hosts: webserver
  vars:
    metadata:
      name: Hello World
      description: This validation prints Hello World!
  roles:
  - hello_world
```

As shown here, the validation playbook requires three top-level directives:

`hosts`, `vars -> metadata` and `roles`

`hosts` specify which nodes to run the validation on.

The `vars` section serves for storing variables that are going to be available to the *Ansible* playbook. The validations API uses the `metadata` section to read validations name and description. These values are then reported by the API.

The validations can be grouped together by specifying a `groups`, a `categories` and a `products` metadata. `groups` are the deployment stage the validations should run on, `categories` are the technical classification for the validations and `products` are the specific validations which should be executed against a specific product.

Groups, Categories and Products function similar to tags and a validation can thus be part of many groups and many categories.

Here is an example:

```
- hosts: webserver
  vars:
    metadata:
      name: Hello World
      description: This validation prints Hello World!
      groups:
        - pre-deployment
        - hardware
      categories:
        - os
        - networking
        - storage
        - security
      products:
        - product1
        - product2
  roles:
  - hello_world
```

property categories

Get the validation list of categories

Returns
A list of categories for the validation

Return type
list or *None* if no metadata has been found

Raise
A *NameError* exception if no metadata has been found in the playbook

Example

```
>>> pl = '/foo/bar/val.yaml'
>>> val = Validation(pl)
>>> print(val.categories)
['category1', 'category2']
```

property get_data

Get the full contents of a validation playbook

Returns
The full content of the playbook

Return type
dict

Example

```
>>> pl = '/foo/bar/val.yaml'
>>> val = Validation(pl)
>>> print(val.get_data)
{'gather_facts': True,
 'hosts': 'all',
 'roles': ['val_role'],
 'vars': {'metadata': {'description': 'description of val ',
                       'groups': ['group1', 'group2'],
                       'categories': ['category1', 'category2'],
                       'products': ['product1', 'product2'],
                       'name': 'validation one'},
                       'var_name1': 'value1'}}
```

property get_formated_data

Get basic information from a validation for output display

Returns
Basic information of a validation including the *Description*, the list of Categories, the list of *Groups*, the *ID* and the *Name*.

Return type
dict

Raise
A *NameError* exception if no metadata has been found in the playbook

Example

```
>>> pl = '/foo/bar/val.yaml'
>>> val = Validation(pl)
>>> print(val.get_formated_data)
{'Categories': ['category1', 'category2'],
 'Products': ['product1', 'product2'],
 'Description': 'description of val',
 'Groups': ['group1', 'group2'],
 'ID': 'val',
 'Name': 'validation one',
 'path': '/tmp/foo/'}
```

property get_id

Get the validation id

Returns
The validation id

Return type
string

Example

```
>>> pl = '/foo/bar/check-cpu.yaml'
>>> val = Validation(pl)
>>> print(val.id)
'check-cpu'
```

`property get_metadata`

Get the metadata of a validation

Returns
The validation metadata

Return type
dict or *None* if no metadata has been found

Raise
A *NameError* exception if no metadata has been found in the playbook

Example

```
>>> pl = '/foo/bar/val1.yaml'
>>> val = Validation(pl)
>>> print(val.get_metadata)
{'description': 'Val1 desc.',
 'groups': ['group1', 'group2'],
 'categories': ['category1', 'category2'],
 'products': ['product1', 'product2'],
 'id': 'val1',
 'name': 'The validation val1's name',
 'path': '/tmp/foo/'}
```

`property get_ordered_dict`

Get the full ordered content of a validation

Returns
An *OrderedDict* with the full data of a validation

Return type
OrderedDict

`property get_vars`

Get only the variables of a validation

Returns
All the variables belonging to a validation

Return type
dict or *None* if no metadata has been found

Raise
A *NameError* exception if no metadata has been found in the playbook

Example

```
>>> pl = '/foo/bar/val.yaml'
>>> val = Validation(pl)
>>> print(val.get_vars)
{'var_name1': 'value1',
 'var_name2': 'value2'}
```

property groups

Get the validation list of groups

Returns
A list of groups for the validation

Return type
list or *None* if no metadata has been found

Raise
A *NameError* exception if no metadata has been found in the playbook

Example

```
>>> pl = '/foo/bar/val.yaml'
>>> val = Validation(pl)
>>> print(val.groups)
['group1', 'group2']
```

property has_metadata_dict

Check the presence of the metadata dictionary

```
- hosts: webserver
  vars:
    metadata:              <====
      name: hello world
      description: this validation prints hello world!
      groups:
        - pre-deployment
        - hardware
      categories:
        - os
        - networking
        - storage
        - security
      products:
        - product1
        - product2
  roles:
  - hello_world
```

Returns
true if *vars* and metadata are found, *false* if not.

Return type
boolean

property has_vars_dict

Check the presence of the vars dictionary

```
- hosts: webserver
  vars:                    <====
    metadata:
      name: hello world
      description: this validation prints hello world!
      groups:
        - pre-deployment
        - hardware
      categories:
        - os
        - networking
        - storage
        - security
      products:
        - product1
        - product2
  roles:
  - hello_world
```

Returns

true if *vars* is found, *false* if not.

Return type

boolean

property products

Get the validation list of products

Returns

A list of products for the validation

Return type

list or *None* if no metadata has been found

Raise

A *NameError* exception if no metadata has been found in the playbook

Example

```
>>> pl = '/foo/bar/val.yaml'
>>> val = Validation(pl)
>>> print(val.products)
['product1', 'product2']
```

validations_libs.validation_actions module

class validations_libs.validation_actions.**ValidationActions**(*validation_path='/usr/share/ansible/vali playbooks'*, *groups_path='/usr/share/ansible/groups.* *log_path='/home/zuul/src/opendev.org/o libs/.tox/docs/validations'*)

Bases: object

An object for encapsulating the Validation Actions

This class allows the possibility to execute the following actions:

- List the available validations
- Show detailed information about one validation
- Show the available parameters for one or multiple validations
- Show the list of the validation groups
- Run one or multiple validations, by name(s) or by group(s)
- Show the history of the validations executions

get_status(*validation_id=None*, *uuid=None*, *status='FAILED'*)

Return validations execution details by status

Parameters

- **validation_id** (string) The validation id
- **uuid** (string) The UUID of the execution
- **status** (string) The status of the execution (Defaults to FAILED)

Returns

A list of validations execution with details and by status

Return type

tuple

Example

```
>>> actions = ValidationActions(validation_path='/foo/bar')
>>> status = actions.get_status(validation_id='foo'))
>>> print(status)
(['name', 'host', 'status', 'task_data'],
 [('Check if debug mode is disabled.',
 'localhost',
 'FAILED',
 {'_ansible_no_log': False,
     'action': 'fail',
     'changed': False,
     'failed': True,
     'msg': 'Debug mode is not disabled.'}),
 ('Check if debug mode is disabled.',
 'localhost',
```

(continues on next page)

(continued from previous page)

```
 'FAILED',
 {'_ansible_no_log': False,
     'action': 'fail',
     'changed': False,
     'failed': True,
     'msg': 'Debug mode is not disabled.'}),
 ('Check if debug mode is disabled.',
 'localhost',
 'FAILED',
 {'_ansible_no_log': False,
     'action': 'fail',
     'changed': False,
     'failed': True,
     'msg': 'Debug mode is not disabled.'})])
```

group_information(*groups=None*, *validation_config=None*)

Get Information about Validation Groups

This is used to print table from python `Tuple` with `PrettyTable`.

```
+----------+--------------------------+-----------------------+
| Groups   | Description              | Number of Validations |
+----------+--------------------------+-----------------------+
| group1   | Description of group1    |                     3 |
| group2   | Description of group2    |                    12 |
| group3   | Description of group3    |                     1 |
+----------+--------------------------+-----------------------+
```

Parameters

- **groups** (`string`) The absolute path of the groups.yaml file. The argument is deprecated and will be removed in the next release. Use the groups_path argument of the init method.
- **validation_config** (`dict`) A dictionary of configuration for Validation loaded from an validation.cfg file.

Returns

The list of the available groups with their description and the numbers of validation belonging to them.

Return type

`tuple`

Example

```
>>> groups = "/foo/bar/groups.yaml"
>>> actions = ValidationActions(constants.ANSIBLE_VALIDATION_DIR,␣
↪groups)
>>> group_info = actions.group_information()
>>> print(group_info)
(('Groups', 'Desciption', 'Number of Validations'),
```

(continues on next page)

(continued from previous page)

```
 [('group1', 'Description of group1', 3),
  ('group2', 'Description of group2', 12),
  ('group3', 'Description of group3', 1)])
```

list_validations(*groups=None*, *categories=None*, *products=None*, *validation_config=None*)

Get a list of the validations selected by group membership or by category. With their names, group membership information, categories and products.

This is used to print table from python `Tuple` with `PrettyTable`.

Parameters

- **groups** (*list*) List of validation groups.
- **categories** (*list*) List of validation categories.
- **products** (*list*) List of validation products.
- **validation_config** (`dict`) A dictionary of configuration for Validation loaded from an validation.cfg file.

Returns

Column names and a list of the selected validations

Return type

tuple

```
-------+-----------+----------------------+---------------+----------
↪----+
| ID    | Name      | Groups               | Categories    | Products␣
↪    |
+------+-----------+----------------------+---------------+----------
↪----+
| val1 | val_name1 | ['group1']           | ['category1'] | [
↪'product1'] |
| val2 | val_name2 | ['group1', 'group2'] | ['category2'] | [
↪'product2'] |
| val3 | val_name3 | ['group4']           | ['category3'] | [
↪'product3'] |
+------+-----------+----------------------+---------------+----------
↪----+
```

Example

```
>>> path = "/foo/bar"
>>> groups = ['group1']
>>> categories = ['category1']
>>> action = ValidationActions(validation_path=path)
>>> results = action.list_validations(groups=groups,
                                      categories=categories)
>>> print(results
```

(continues on next page)

(continued from previous page)

```
(('ID', 'Name', 'Groups', 'Categories', 'Products'),
 [('val1',
   'val_name1',
   ['group1'],
   ['category1'],
   ['product1']),
  ('val2',
   'val_name2',
   ['group1', 'group2'],
   ['category2'],
   ['product2'])])
```

run_validations(*validation_name=None*, *inventory='localhost'*, *group=None*, *category=None*, *product=None*, *extra_vars=None*, *validations_dir=None*, *extra_env_vars=None*, *ansible_cfg=None*, *quiet=True*, *limit_hosts=None*, *run_async=False*, *base_dir='/usr/share/ansible'*, *python_interpreter=None*, *skip_list=None*, *callback_whitelist=None*, *output_callback='vf_validation_stdout'*, *ssh_user=None*, *validation_config=None*, *exclude_validation=None*, *exclude_group=None*, *exclude_category=None*, *exclude_product=None*)

Run one or multiple validations by name(s), by group(s) or by product(s)

Parameters

- **validation_name** (`list`) A list of validation names.
- **inventory** (`string`) Either proper inventory file, or a comma-separated list. (Defaults to `localhost`)
- **group** (`list`) A list of group names
- **category** (`list`) A list of category names
- **product** (`list`) A list of product names
- **extra_vars** (*Either a Dict or the absolute path of JSON or YAML*) Set additional variables as a Dict or the absolute path of a JSON or YAML file type.
- **validations_dir** (`string`) The absolute path of the validations playbooks
- **extra_env_vars** (`dict`) Set additional ansible variables using an extravar dictionary.
- **ansible_cfg** (`string`) Path to an ansible configuration file. One will be generated in the artifact path if this option is None.
- **quiet** (`Boolean`) Disable all output (Defaults to `True`)
- **limit_hosts** (`string`) Limit the execution to the hosts.
- **run_async** (`boolean`) Enable the Ansible asynchronous mode (Defaults to `False`)
- **base_dir** (`string`) The absolute path of the validations base directory (Defaults to `constants.DEFAULT_VALIDATIONS_BASEDIR`)

- **python_interpreter** (string) Path to the Python interpreter to be used for module execution on remote targets, or an automatic discovery mode (auto, auto_silent or the default one auto_legacy)
- **callback_whitelist** (list or string) Comma separated list of callback plugins. Custom output_callback is also whitelisted. (Defaults to None)
- **output_callback** (string) The Callback plugin to use. (Defaults to validation_stdout)
- **skip_list** (dict) List of validations to skip during the Run form as {xyz: {hosts: ALL, reason: None, lp: None} } (Defaults to None)
- **ssh_user** (string) Ssh user for Ansible remote connection
- **validation_config** (dict) A dictionary of configuration for Validation loaded from an validation.cfg file.
- **exclude_validation** (*list*) List of validation name(s) to exclude
- **exclude_group** (*list*) List of validation group(s) to exclude
- **exclude_category** (*list*) List of validation category(s) to exclude
- **exclude_product** (*list*) List of validation product(s) to exclude

Returns

A list of dictionary containing the informations of the validations executions (Validations, Duration, Host_Group, Status, Status_by_Host, UUID and Unreachable_Hosts)

Return type

list

Raises

ValidationRunException

Example

```
>>> path = "/u/s/a"
>>> validation_name = ['foo', 'bar']
>>> actions = ValidationActions(validation_path=path)
>>> results = actions.run_validations(inventory='localhost',
                                      validation_name=validation_
↪name,
                                      quiet=True)
>>> print(results)
[{'Duration': '0:00:02.285',
  'Host_Group': 'all',
  'Status': 'PASSED',
  'Status_by_Host': 'localhost,PASSED',
  'UUID': '62d4d54c-7cce-4f38-9091-292cf49268d7',
  'Unreachable_Hosts': '',
  'Validations': 'foo'},
 {'Duration': '0:00:02.237',
  'Host_Group': 'all',
  'Status': 'PASSED',
```

(continues on next page)

(continued from previous page)

```
 'Status_by_Host': 'localhost,PASSED',
 'UUID': '04e6165c-7c33-4881-bac7-73ff3f909c24',
 'Unreachable_Hosts': '',
 'Validations': 'bar'}]
```

show_history(*validation_ids=None*, *extension='json'*, *history_limit=None*)

Return validation executions history

Parameters

- **validation_ids** (*a list of strings*) The validation ids
- **extension** (*string*) The log file extension (Defaults to `json`)
- **history_limit** (*int*) The number of most recent history logs to be displayed.

Returns

Returns the information about the validation executions history

Return type

`tuple`

Example

```
>>> actions = ValidationActions(constants.ANSIBLE_VALIDATION_DIR)
>>> print(actions.show_history())
(('UUID', 'Validations', 'Status', 'Execution at', 'Duration'),
 [('5afb1597-e2a1-4635-b2df-7afe21d00de6',
 'foo',
 'PASSED',
 '2020-11-13T11:47:04.740442Z',
 '0:00:02.388'),
 ('32a5e217-d7a9-49a5-9838-19e5f9b82a77',
 'foo2',
 'PASSED',
 '2020-11-13T11:47:07.931184Z',
 '0:00:02.455'),
 ('62d4d54c-7cce-4f38-9091-292cf49268d7',
 'foo',
 'PASSED',
 '2020-11-13T11:47:47.188876Z',
 '0:00:02.285'),
 ('04e6165c-7c33-4881-bac7-73ff3f909c24',
 'foo3',
 'PASSED',
 '2020-11-13T11:47:50.279662Z',
 '0:00:02.237')])
>>> actions = ValidationActions(constants.ANSIBLE_VALIDATION_DIR)
>>> print(actions.show_history(validation_ids=['foo']))
(('UUID', 'Validations', 'Status', 'Execution at', 'Duration'),
 [('5afb1597-e2a1-4635-b2df-7afe21d00de6',
 'foo',
```

(continues on next page)

(continued from previous page)

```
 'PASSED',
 '2020-11-13T11:47:04.740442Z',
 '0:00:02.388'),
 ('04e6165c-7c33-4881-bac7-73ff3f909c24',
 'foo',
 'PASSED',
 '2020-11-13T11:47:50.279662Z',
 '0:00:02.237')])
```

show_validations(*validation*, *validation_config=None*)

Display detailed information about a Validation

Parameters

- **validation** (*string*) The name of the validation
- **validation_config** (dict) A dictionary of configuration for Validation loaded from an validation.cfg file.

Returns

The detailed information for a validation

Return type

dict

Raises

ValidationShowException

Example

```
>>> path = "/foo/bar"
>>> validation = 'foo'
>>> action = ValidationActions(validation_path=path)
>>> results = action.show_validations(validation=validation)
>>> print(results)
{
 'Description': 'Description of the foo validation',
 'Categories': ['category1', 'category2'],
 'Groups': ['group1', 'group2'],
 'ID': 'foo',
 'Last execution date': None,
 'Name': 'Name of the validation foo',
 'Number of execution': 'Total: 0, Passed: 0, Failed: 0',
 'Parameters': {'foo1': bar1}
}
```

show_validations_parameters(*validations=None*, *groups=None*, *categories=None*, *products=None*, *output_format='json'*, *download_file=None*, *validation_config=None*)

Return Validations Parameters for one or several validations by their names, their groups, by their categories or by their products.

Parameters

- **validations** (*list*) List of validation name(s)
- **groups** (*list*) List of validation group(s)
- **categories** (*list*) List of validation category(ies)
- **products** (*list*) List of validation product(s)
- **output_format** (*string*) Output format (Supported format are JSON or YAML)
- **download_file** (*string*) Path of a file in which the parameters will be stored
- **validation_config** (`dict`) A dictionary of configuration for Validation loaded from an validation.cfg file.

Returns

A JSON or a YAML dump (By default, JSON). if *download_file* is used, a file containing only the parameters will be created in the file system.

Raises

ValidationShowException

Example

```
>>> validations = ['check-cpu', 'check-ram']
>>> groups = None
>>> categories = None
>>> products = None
>>> output_format = 'json'
>>> show_validations_parameters(validations, groups,
                                categories, products, output_format)
{
    "check-cpu": {
        "parameters": {
            "minimal_cpu_count": 8
        }
    },
    "check-ram": {
        "parameters": {
            "minimal_ram_gb": 24
        }
    }
}
```

validations_libs.validation_logs module

class validations_libs.validation_logs.**ValidationLog**(*uuid=None*, *validation_id=None*, *logfile=None*, *log_path='/home/zuul/src/opendev.org/openstack/validations-libs/.tox/docs/validations'*, *extension='json'*)

Bases: `object`

An object for encapsulating a Validation Log file

property get_duration

Return duration of Ansible runtime

Return type
string

property get_host_group

Return host group

Returns
A comma-separated list of host(s)

Return type
string

property get_hosts_status

Return status by host(s)

Returns
A comma-separated string of host with its status

Return type
string

Example

```
>>> logfile = '/tmp/123_foo_2020-03-30T13:17:22.447857Z.json'
>>> val = ValidationLog(logfile=logfile)
>>> print(val.get_hosts_status)
'localhost,PASSED, webserver1,FAILED, webserver2,PASSED'
```

get_log_path()

Return full path of a validation log

property get_logfile_content

Return logfile content

Return type
dict

property get_logfile_datetime

Return log file datetime from a UUID and a validation ID

Returns
The datetime of the log file

Return type
list

Example

```
>>> logfile = '/tmp/123_foo_2020-03-30T13:17:22.447857Z.json'
>>> val = ValidationLog(logfile=logfile)
>>> print(val.get_logfile_datetime)
['2020-03-30T13:17:22.447857Z']
```

property get_logfile_infos

Return log file information from the log file basename

Returns
A list with the UUID, the validation name and the datetime of the log file

Return type
`list`

Example

```
>>> logfile = '/tmp/123_foo_2020-03-30T13:17:22.447857Z.json'
>>> val = ValidationLog(logfile=logfile)
>>> print(val.get_logfile_infos)
['123', 'foo', '2020-03-30T13:17:22.447857Z']
```

property get_plays

Return a list of Playbook data

property get_reason

Return validation reason

Returns
hostname: reason of the failure

Return type
`string`

property get_start_time

Return Ansible start time

Return type
`string`

property get_status

Return validation status

Returns
FAILED if there are any failed or unreachable validations, SKIPPED if skipped is True and ok is false which means that the entire validation has been ignored because no host matched, PASSED if none of those conditions.

Return type
`string`

property get_tasks_data

Return a list of task from validation output

property get_unreachable_hosts

Return unreachable hosts

Returns
A list of unreachable host(s)

Return type
`string`

Example

- Multiple unreachable hosts

```
>>> logfile = '/tmp/123_foo_2020-03-30T13:17:22.447857Z.json'
>>> val = ValidationLog(logfile=logfile)
>>> print(val.get_unreachable_hosts)
'localhost, webserver2'
```

- Only one unreachable host

```
>>> logfile = '/tmp/123_foo_2020-03-30T13:17:22.447857Z.json'
>>> val = ValidationLog(logfile=logfile)
>>> print(val.get_unreachable_hosts)
'localhost'
```

- No unreachable host

```
>>> logfile = '/tmp/123_foo_2020-03-30T13:17:22.447857Z.json'
>>> val = ValidationLog(logfile=logfile)
>>> print(val.get_unreachable_hosts)
''
```

property get_uuid

Return log uuid

Return type
`string`

property get_validation_id

Return validation id

Return type
`string`

is_valid_format()

Return True if the log file is a valid validation format

The validation log file has to contain three level of data.

- `plays` will contain the Ansible execution logs of the playbooks
- `stat` will contain the statistics for each targeted hosts
- `validation_output` will contain only the warning or failed tasks

```
{
  'plays': [],
  'stats': {},
  'validation_output': []
}
```

Returns
`True` if the log file is valid, `False` if not.

Return type
boolean

class validations_libs.validation_logs.**ValidationLogs**(*logs_path='/home/zuul/src/opendev.org/openstac libs/.tox/docs/validations'*)

Bases: object

An object for encapsulating the Validation Log files

get_all_logfiles(*extension='json'*)

Return logfiles from logs_path

Parameters
extension (string) The extension file (Defaults to json)

Returns
A list of the absolute path log files

Return type
list

get_all_logfiles_content()

Return logfiles content

Returns
A list of the contents of every log files available

Return type
list

get_logfile_by_uuid(*uuid*)

Return logfiles by uuid

Parameters
uuid (string) The UUID of the validation execution

Returns
The list of the log files by UUID

Return type
list

get_logfile_by_uuid_validation_id(*uuid*, *validation_id*)

Return logfiles by uuid and validation_id

Parameters

- **uuid** (string) The UUID of the validation execution
- **validation_id** (string) The ID of the validation

Returns
A list of the log files by UUID and validation_id

Return type
list

get_logfile_by_validation(*validation_id*)

Return logfiles by validation_id

Parameters
validation_id (string) The ID of the validation

Returns
The list of the log files for a validation

Return type
list

get_logfile_content_by_uuid(*uuid*)
Return logfiles content by uuid

Parameters
uuid (string) The UUID of the validation execution

Returns
The list of the log files contents by UUID

Return type
list

get_logfile_content_by_uuid_validation_id(*uuid*, *validation_id*)
Return logfiles content filter by uuid and validation_id

Parameters

- **uuid** (string) The UUID of the validation execution
- **validation_id** (string) The ID of the validation

Returns
A list of the log files content by UUID and validation_id

Return type
list

get_logfile_content_by_validation(*validation_id*)
Return logfiles content by validation_id

Parameters
validation_id (string) The ID of the validation

Returns
The list of the log files contents for a validation

Return type
list

get_results(*uuid*, *validation_id=None*)
Return a list of validation results by uuid Can be filter by validation_id

Parameters

- **uuid** (string` or ``list) The UUID of the validation execution
- **validation_id** (string) The ID of the validation

Returns
A list of the log files content by UUID and validation_id

Return type
list

Example

```
>>> v_logs = ValidationLogs()
>>> uuid = '78df1c3f-dfc3-4a1f-929e-f51762e67700'
>>> print(v_logs.get_results(uuid=uuid)
[{'Duration': '0:00:00.514',
  'Host_Group': 'undercloud,Controller',
  'Status': 'FAILED',
  'Status_by_Host': 'undercloud,FAILED, undercloud,FAILED',
  'UUID': '78df1c3f-dfc3-4a1f-929e-f51762e67700',
  'Unreachable_Hosts': 'undercloud',
  'Validations': 'check-cpu'}]
```

`get_validations_stats`(*logs*)

Return validations stats from log files

Parameters

logs (`list`) A list of log file contents

Returns

Information about validation statistics. `last execution date` and `number of execution`

Return type

`dict`

Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

P

R

S

T

V

W