Teflo Documentation

Release 2.4.0

Red Hat Inc.

Apr 06, 2023

INSTALLATION CONFIGURATION

Warning: This project is in maintenance mode and will not have any new feature development.

CHAPTER

ONE

WHAT IS TEFLO?

TEFLO stands for (**T** est **E** xecution **F** ramework **L** ibraries and **O** bjects)

Teflo is an orchestration software that controls the flow of a set of testing scenarios. It is a standalone tool that includes all aspects of the workflow. It allows users to provision machines, deploy software, execute tests against them and manage generated artifacts and report results.

Teflo Provides *structure*, *readability*, *extensibility* and *flexibility* by :

- providing a DSL (YAML) to express a test workflow as a series of steps.
- enabling integration of external tooling to execute the test workflow as defined by the steps.

Teflo can be used for an E2E (end to end) multi-product scenario. Teflo handles coordinating the E2E task workflow to drive the scenario execution.

CHAPTER TWO

WHAT DOES AN E2E WORKFLOW CONSIST OF?

At a high level teflo executes the following tasks when processing a scenario.

- Provision system resources
- Perform system configuration
- Install products
- Configure products
- Install test frameworks
- Configure test frameworks
- Execute tests
- Report results
- Destroy system resources
- Send Notifications

Teflo is a test execution framework. It is a standalone tool written in Python. Teflo can perform the following tasks

Provision - Create resources they want to test on (physical resources, VMs etc)

Orchestrate - Configure these resources , like install packages on them, run scripts, ansible playbooks etc

Execute - Execute actual tests on the configured resources

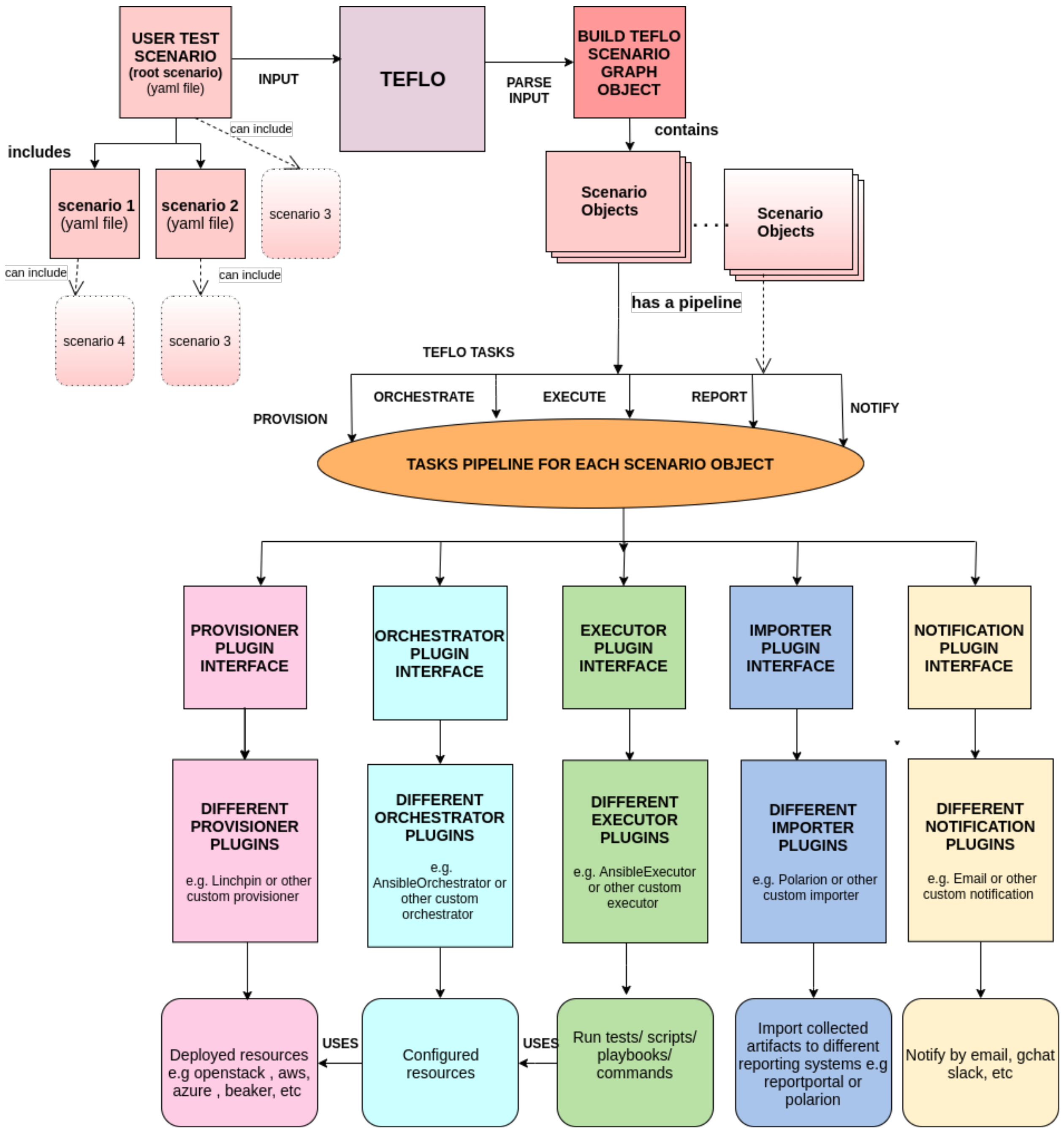
Report - Send or collect logs from the run tests

Notification - Send email/gchat/slack notification during each stage of teflo run or at the end based on the triggers set

Cleanup - Cleanup all the deployed resources.

These tasks can be run individually or together.

Teflo follows a pluggable architechture, where users can add different pluggins to support external tools Below is a diagram that gives you a quick overview of Teflo workflow


USER TEST SCENARIO (root scenario) (yaml file)
INPUT
TEFLO
PARSE INPUT
BUILD TEFLO SCENARIO GRAPH OBJECT
contains
includes
can include
scenario 1 (yaml file)
scenario 2 (yaml file)
scenario 3
can include
can include
scenario 4
scenario 3
Scenario Objects
Scenario Objects
has a pipeline
TEFLO TASKS
PROVISION
ORCHESTRATE
EXECUTE
REPORT
NOTIFY
TASKS PIPELINE FOR EACH SCENARIO OBJECT
PROVISIONER PLUGIN INTERFACE
ORCHESTRATOR PLUGIN INTERFACE
EXECUTOR PLUGIN INTERFACE
IMPORTER PLUGIN INTERFACE
NOTIFICATION PLUGIN INTERFACE
DIFFERENT PROVISIONER PLUGINS e.g. Linchpin or other custom provisioner
DIFFERENT ORCHESTRATOR PLUGINS e.g. AnsibleOrchestrator or other custom orchestrator
DIFFERENT EXECUTOR PLUGINS e.g. AnsibleExecutor or other custom executor
DIFFERENT IMPORTER PLUGINS e.g. Polarion or other custom importer
DIFFERENT NOTIFICATION PLUGINS e.g. Email or other custom notification
Deployed resources e.g openstack , aws, azure , beaker, etc
USES
Configured resources
USES
Run tests/ scripts/ playbooks/ commands
Import collected artifacts to different reporting systems e.g reportportal or polarion
Notify by email, gchat slack, etc

- To learn more about how to set up and use Teflo please check out the Users Guide
- To know how to create a custom plugin checkout Developers Guide
- To know about our release cadence and contribution policy check out Release Cadence

2.1 Install Teflo

2.1.1 Requirements

Your system requires the following packages to install teflo:

```
# To install git using dnf package manager
$ sudo dnf install -y git

# To install git using yum package manager
$ sudo yum install -y git

# Install python pip: https://pip.pypa.io/en/stable/installing
$ sudo curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
$ sudo python get-pip.py

# Recommend installation of virtualenv using pip
$ sudo pip install virtualenv
```

Note: To install Teflo pip version 18.1 or higher is required

2.1.2 Install

Install teflo from source:

```
# for ansible modules requiring selinux, you will need to enable system site packages
$ virtualenv --system-site-packages teflo
$ source teflo/bin/activate
(teflo) $ pip install teflo
```

2.1.3 Post Install

If you require teflo to interface with beaker using the bkr-client provisioner, you will need to enable the beaker client repository/install the beaker-client package. Teflo uses the beaker client package to provision physical machines in beaker.

```
# https://beaker-project.org/download.html
$ sudo curl -o /etc/yum.repos.d/bkr-client.repo \
https://beaker-project.org/yum/beaker-client-<DISTRO>.repo

# To install beaker-client using dnf package manager
$ sudo dnf install -y beaker-client

# To install beaker-client using yum package manager
$ sudo yum install -y beaker-client
```

Note: Beaker-client could be installed from PyPI rather than RPM. Installing from pip fails in Python 3. Beaker client is not compatible with Python 3 currently. Once compatibile it can be installed with teflo. Teflo is Python 3 compatible.

2.1.4 Teflo External Plugin Requirements

Teflo is able to use external tools using its plugins. These plugins need to be installed separately.

Users can develop Teflo has plugins for provisioners, orchestrators, executors, importers and notifiers. Following are the plugins currently developed and supported by Teflo

Provisioner Plugins

Teflo_Linchpin_Plugin

This plugin can be use to provision using the Linchpin tool. The Linchpin plugin will be available as an extra. To install Linchpin certain requirements need to be met so that it can be installed correctly. Please refer to the before install section of the plugin documentation on how to install them.

Once installed, you can install Linchpin from Teflo

```
$ pip install teflo[linchpin-wrapper]
```

Once Linchpin_Plugin is installed, you will get support for all providers that linchpin supports. Although there are some providers that require a few more dependencies to be installed. Refer to the post-install section of the plugin document for methods on how to install those dependencies.

Openstack_Client_Plugin

This plugin is used to Provision openstack assets using openstack-client tool This plugin is also available as extra. To install this plugin do the following Refer here to get more information on how to use the plugin

```
$ pip install teflo[openstack-client-plugin]
```

Importer Plugins

Teflo_Polarion_Plugin

This plugin allows teflo to send test results to Polarion tool. This plugin allows teflo to import xunit files to Polarion by using the Polar library. Polar library helps converts the generic xUnit file by applying Polarion specific tags and import them to Polarion and monitor their progress teflo_polarion_plugin uses the parameters declared in the Teflo's Scenario Descriptor File Report section to send the xunit files to Polarion

Note: This plugin is meant for Internal RED HAT use and is not available publicly yet

Teflo_Rppreproc_Plugin

This plugin allows teflo to send test results to Report Portal tool. Based on the input provided by Teflo's Scenario Descriptor File (SDF),the teflo_rppreproc_plugin validates the config file for report portal client if provided else creates one using the other parameters in the SDF, creates appropriate payload (logs and attachements)for the report portal client and uses Teflo's helper methods to send the payload to the report portal client by running the rp_preproc commands

Note: This plugin is meant for Internal RED HAT use and is not available publicly yet

Teflo_Terraform_Plugin

This plugin is used to call terraform as a provisioner Please review the repo documentation

```
$ pip install teflo[terraform-plugin]
```

Notification Plugins

Teflo_Webhooks_Notification_Plugin

This plugin is used to notify based users using chat applications gchat and slack. Please review the repo documentation and how to use the plugin. Please review Teflo's notification triggers to get more info on using Teflo`s notification feature

```
$ pip install teflo[webhook-notification-plugin]
```

Teflo_Notify_Service_Plugin

This plugin is used to notify based users using chat applications gchat and slack. Please review the repo documentation and how to use the plugin.Please review Teflo's notification triggers to get more info on using Teflo`s notification feature

```
$ pip install teflo[notify-service-plugin]
```

2.1.5 Teflo Matrix for Plugins

The table below lists out the released Teflo version and supported teflo plugin versions. This matrix will track n and n-2 teflo releases

Table 1: Teflo plugin matrix for n and n-2 releases

Teflo Release	2.2.7	2.2.8	2.2.9	2.3.0	2.4.0
Rppreproc Plugin	2.0.2	2.0.2	2.0.2	2.0.2	2.0.2
Polarion Plugin	1.1.0	1.1.0	1.1.0	1.1.0	1.1.0
Linchpin Plugin	1.0.2	1.0.2	Depreciated	Depreciated	Depreciated
Openstack Client Plugin	1.0.1	1.0.1	1.0.1	1.0.1	1.0.2
Webhooks Notification Plugin	2.0.1	2.0.1	2.0.1	2.0.1	2.0.1
Terraform Plugin	1.0.0	1.0.0	1.0.0	1.0.0	1.0.1
Notify Service Plugin	2.0.1	2.0.1	2.0.1	2.0.1	2.0.1
Polar	1.2.2	1.2.2	1.2.2	1.2.2	1.2.2
Rp_preproc	0.3.1	0.3.2	0.3.4	0.3.5	0.3.6

2.2 Configure Teflo

This is a mandatory configuration file, where you set your credentials, and there are many optional settings to help you adjust your usage of Teflo. The credentials of the configuration file is the only thing that is mandatory. Most of the other default configuration settings should be sufficient; however, please read through the options you have.

Where it is loaded from (using precedence low to high):

1. /etc/teflo/teflo.cfg
2. ./teflo.cfg (current working directory)
3. TEFLO_SETTINGS environment variable to the location of the file

Important: It is important to realize if you have a configuration file set using both options, the configuration files will be combined, and common key values will be overridden by the higher precedent option, which will be the TEFLO_SETTINGS environment variable.

Configuration example (with all options):

```
# teflo config file
# ==================

# the config file provides an additional way to define teflo parameters

# config file is searched for in the following order below. a configuration
# setting will be overrided if another source is found last
#   1. /etc/teflo/teflo.cfg
#   2. ./teflo.cfg (current working directory)
#   3. TEFLO_SETTINGS (environment variable)

# default settings

[defaults]
log_level=debug
# Path for teflo's data folder where teflo logs will be stored
data_folder=/var/local/teflo
workspace=.
```

(continues on next page)

(continued from previous page)

```
# Endpoint URL of Cachet Status page
# Cachet status page.
resource_check_endpoint=<endpoint_ url_for_dependency_check>
# The teflo run exits on occurrence of a failure of a task in a scenario, if a user␣
↪wants to continue
# the teflo run, in spite of one task failure, the skip_fail parameter can be set to␣
↪true in
# the teflo.cfg or passed using cli.
skip_fail=False
#
# A static inventory path can be used for ansible inventory file.
# Can be relative path in teflo scenario workspace
# inventory_folder=static/inventory
#
# Can be a directory in the user $HOME path
# inventory_folder=~/scenario/static/inventory
#
# Can be an absolute path
# inventory_folder=/test/scenario/static/inventory
#
# Can be a path containing an environment variable
# inventory_folder=$WORKSPACE/scenario/static/inventory
# default value of the inventory folder is 'TEFLO_DATA_FOLDER/.results/inventory'
inventory_folder=<path for ansible inventory files>
# credentials file and Vault password
# User can set all teh credential information in a text file and encrypt it using␣
↪ansible vault
# provide the path in under CREDENTIALS_PATH. Provide the vault password here. This␣
↪password can be
# exported as an environmental variable
CREDENTIAL_PATH=<path for the credetials txt file>
VAULTPASS=<ansible vault password>


# time out for each stage
# you can set the timeout value for each of teflo's stages (validation, provision,␣
↪orchestrate, execute, report and cleanup)
[timeout]
provision=500
cleanup=1000
orchestrate=300
execute=200
report=100
validate=10


# credentials settings

[credentials:beaker-creds]
hub_url=<hub_url>
keytab=<keytab>
keytab_principal=<keytab_principal>
```

(continues on next page)

(continued from previous page)

```
username=<username>
password=<password>

[credentials:openstack-creds]
auth_url=<auth_url>
tenant_name=<tenant_name>
username=<username>
password=<password>
domain_name=<domain_name>

# orchestrator settings

[orchestrator:ansible]
# remove ansible log
log_remove=False
# set the verbosity
# this option will override the max verbosity when log level is set to debug.
verbosity=vv

[task_concurrency]
# this controls how tasks (provision, orchestrate, execute, report) are executed
# by Teflo either parallel or sequential.
# When set to False the task will execute sequentially.
provision=False


# executor settings

[executor:runner]
# set the testrun_results to false if you dont want it to be collected in the logs for␣
↪the xml files collected during
# execution
testrun_results=False
# Teflo by default will NOT exit if the collection of artifact task fails. In order to␣
↪exit the run on an error during
# collection of artifacts user can set this field to true , else False or ignore the␣
↪field.
exit_on_error=True

# Teflo Alias
[alias]
dev_run=run -s scenario.yml --log-level debug --iterate-method by_depth
prod_run=show -s scenario.yml --list-labels
```

Note: Many of the configuration options can be overridden by passing cli options when running teflo. See the options in the running teflo example.

2.2.1 Using Jinja Variable Data

Teflo uses Jinja2 template engine to be able to template variables within the teflo.cfg file. Teflo allows template variable data to be set as environmental variables

Here is an example teflo.cfg file using Jinja to template some variable data:

```
[credentials:openstack]
auth_url=<auth_url>
username={{ OS_USER }}
password={{ OS_PASSWORD }}
tenant_name={{ OS_TENANT }}
domain_name=redhat.com

[task_concurrency]
provision=True
report=False
orchestrate={{ ORC_TASK_CONCURRENCY }}
```

Prior to running teflo, the templated variables will have to be exported

```
export OS_USER=user1
export OS_PASSWORD=password
export OS_TENANT=project1
export ORC_TASK_CONCURRENCY=True
```

2.2.2 inventory_folder

The **inventory_folder** option is not a required option but it is important enough to note its usage. By default teflo will create an inventory directory containing ansible inventory files in its data directory. These are used during orchestration and execution. Refer to the Teflo Output page.

Some times this is not desired behavior. This option allows a user to specify a static known directory that Teflo can use to place the ansible inventory files. If the specified directory does not exist, teflo will create it and place the ansible inventory files. If it does, teflo will only place the ansible files in the directory. Teflo will then use this static directory during orchestrate and execution.

2.2.3 task_concurrency

The **task_concurrency** option is used to control how tasks are executed by Teflo. Whether it should be sequential or in parallel/concurrent. Below is the default execution type of each of the Teflo tasks:

Key	Concurrent	Type
validate	True	String
provision	True	String
orchestrate	False	String
execute	False	String
report	False	String

There are cases where it makes sense to adjust the execution type. Below are some examples:

There are cases when provisioning assets of different types that there might be an inter-dependency so executing the tasks in parallel will not suffice, i.e. provision a virtual network and a VM attached to that network. In that case, set the **provision=False** and arrange the assets in the scenario descriptor file in the proper sequential order.

There are cases when you need to import the same test artifact into separate reporting systems but one reporting systems needs the data in the test artifact to be modified with metadata before it can be imported. i.e modify and import into Polarion with Polarion metadata and then import that same artifact into Report Portal. In that case, set the **report=False** and arrange the resources defined in the scenario descriptor file in the proper sequential order.

There could be a case where you would like to execute two different test suites concurrently because they have no dependency on each other or there is no affect to each other. In that case, set the **execute=True** to have them running concurrently.

2.3 User's Guide

2.3.1 Teflo Quickstart

Welcome to the teflo quick start guide! This guide will help get you started with using teflo. This guide is broken down into two sections:

1. Teflo Usage
2. Getting Started Examples

Teflo usage will provide you with an overview of how you can call teflo. It can be called from either a command line or invoked within a Python module. The getting started examples section will show you working examples for each teflo task. Each example is stored within a git repository for you to clone and try in your local environment.

Note: At this point, you should already have teflo installed and configured. If not, please view the install guide and the configuration guide.

Teflo Usage

Once teflo is installed, you can run the teflo command to view its options:

```
# OUTPUT MAY VARY BETWEEN RELEASES

$ teflo
Usage: teflo [OPTIONS] COMMAND [ARGS]...

  Teflo - (Test Execution Framework Libraries and Objects)

  It is an orchestration software that controls the flow of a set of testing
  scenarios.

  It was formerly known as Carbon

Options:
  -v, --verbose  Add verbosity to the commands.
  --version      Show the version and exit.
```

(continues on next page)

(continued from previous page)

```
  --help         Show this message and exit.

Commands:
  alias     Run predefined command from teflo.cfg
  init      Initializes a teflo project in your workspace.
  notify    Trigger notifications marked on demand for a scenario.
  run       Run a scenario configuration.
  show      Show information about the scenario.
  validate  Validate a scenario configuration.
```

Run

The run command will run your scenario descriptor executing all tasks you select. Below are the available run command options.

```
# OUTPUT MAY VARY BETWEEN RELEASES

$ teflo run --help
Usage: teflo run [OPTIONS]

  Run a scenario configuration.

Options:
  -t, --task [validate|provision|orchestrate|execute|report|cleanup]
                                  Select task to run. (default=all)
  -s, --scenario                  Scenario definition file to be executed.
  -d, --data-folder               Directory for saving teflo runtime files.
  -w, --workspace                 Scenario workspace.
  --log-level [debug|info]        Select logging level. (default=info)
  --vars-data                     Pass in variable data to template the
                                  scenario. Can be a file or raw json.

  -l, --labels                    Use only the resources associated with
                                  labels for running the tasks. labels and
                                  skip_labels are mutually exclusive

  -sl, --skip-labels              Skip the resources associated with
                                  skip_labels for running the tasks. labels
                                  and skip_labels are mutually exclusive

  -sn, --skip-notify              Skip triggering the specific notification
                                  defined for the scenario.

  -nn, --no-notify                Disable sending an notifications defined for
                                  the scenario.

  -sf, --skip-fail                The teflo run exits on occurrence of a failure of a␣
↪task in
                                  a scenario, if user wants to continue the teflo run,␣
↪in spite
```

(continues on next page)

(continued from previous page)

```
                                of one task failure, the skip_fail flag will allow it.

  --help                        Show this message and exit.
```

Running Included Scenarios

With Teflo Version 2.0 onwards , Teflo supports recursive inclusion of scenarios, i.e. a parent scenario can have more than one included scenarios, and these included scenarios then can have more included scenarios. This is handled by Teflo using a Scenario Graph data structure. Please view Included Scenarios to know more.

During a teflo run , based on what tasks are to be run, a task pipeline is created for each scenario. These pipelines are run sequentially in the order of how the scenario_graph is traversed. Within each pipeline an individual task can be run sequentially or concurrently as before. Please view Scenario Graph to understand how included scenarios will be executed.

For .e.g. if the tasks to be done are provision and orchestrate and included scenarios are being used, then based on how the scenario graph is traversed, the provision and orchestrate pipeline will be run (sequentially or concurrently based on the settings in teflo.cfg) for each scenario in the graph.

The exception to this rule are the validate and cleanup task, for which the entire scenario graph is considered together and validated.

Note: For version 1.2.5 and below

If 'Include' section is present in the scenario file, teflo will aggregate and execute the selected tasks from both, main/parent and the included scenario file. e.g. if common.yml is the included scenario file, scenario.yml is the main scenario file and task selected is provision,the provision pipeline is created with provision tasks from included scenario followed by the provision tasks from main scenario.

Note: There is no separate cleanup section within the scenario descriptor file (SDF). When the cleanup task is run, Teflo looks for if any assets/resources are provisioned, and if so it will destroy them Also the cleanup task will look for orchestrate tasks in the SDF with the keyword *cleanup* defined and run any scripts/playbooks mentioned there as a part of cleanup process. Example for orchestrate task cleanup

Option	Description	Require	Default
task	Defines which teflo task to execute the scenario against.	No	All tasks
scenario	This is the scenario descriptor filename. It can be either a relative or absoluate path to the file.	Yes	N/A
data-folde	The data folder is where all teflo runs are stored. Every teflo run will create a unique folder for that run to store its output. By default teflo uses /tmp as the data folder to create sub folders for each run. You can override this to define the base data folder.	No	/tmp
work	The scenario workspace is the directory where your scenario exists. Inside this directory is all the necessary files to run the scenario.	No	./ (current working directory)
log-level	The log level defines the logging level for messages to be logged.	No	Info
skip-fail	The teflo run exits on occurrence of a failure of a task in a scenario, if user wants to continue the teflo run, in spite of one task failure, the skip_fail parameter can be set to true in the teflo.cfg or passed using cli.	No	False

To run your scenario executing all given tasks, run the following command:

```
$ teflo run --scenario <scenario>
```

```
from yaml import safe_load
from teflo import Teflo

cbn = Teflo('teflo')

with open('<scenario>, 'r') as f:
    cbn.load_from_yaml(list(safe_load(f)))

cbn.run()
```

You have the ability to only run a selected task. You can do this by the following command:

```
# individual task
$ teflo run --scenario <scenario> --task <task>

# multiple tasks
$ teflo run --scenario <scenario> --task <task> --task <task>
```

```
from yaml import safe_load
from teflo import Teflo

cbn = Teflo('teflo')

with open('<scenario>, 'r') as f:
    cbn.load_from_yaml(list(safe_load(f)))

# individual task
cbn.run(tasklist=['task'])

# multiple tasks
```

(continues on next page)

(continued from previous page)

```
cbn.run(tasklist=['task', 'task'])
```

Validate

The validate command validates the scenario descriptor.

```
$ teflo validate --help
Usage: teflo validate [OPTIONS]

  Validate a scenario configuration.

Options:
  -t, --task [validate|provision|orchestrate|execute|report|cleanup]
                                  Select task to run. (default=all)
  -s, --scenario                  Scenario definition file to be executed.
  -d, --data-folder               Directory for saving teflo runtime files.
  -w, --workspace                 Scenario workspace.
  --log-level [debug|info]        Select logging level. (default=info)
  --vars-data                     Pass in variable data to template the
                                  scenario. Can be a file or raw json.
  -l, --labels                    Use only the resources associated with
                                  labels for running the tasks. labels and
                                  skip_labels are mutually exclusive
  -sl, --skip-labels              Skip the resources associated with
                                  skip_labels for running the tasks. labels
                                  and skip_labels are mutually exclusive
  -sn, --skip-notify              Skip triggering the specific notification
                                  defined for the scenario.
  -nn, --no-notify                Disable sending any notifications defined for
                                  the scenario.
  --help                          Show this message and exit.
```

Notify

Trigger notifications marked on demand for a scenario configuration.

This is useful when there is a break in the workflow, between when the scenario completes and the triggering of the notification.

```
teflo notify --help
Usage: teflo notify [OPTIONS]

    Trigger notifications marked on demand for a scenario configuration.

Options:
    -s, --scenario              Scenario definition file to be executed.
    -d, --data-folder           Directory for saving teflo runtime files.
    -w, --workspace             Scenario workspace.
    --log-level [debug|info]    Select logging level. (default=info)
    --vars-data                 Pass in variable data to template the scenario.
```

(continues on next page)

(continued from previous page)

```
                              Can be a file or raw json.
    -sn, --skip-notify        Skip triggering the specific notification
                              defined for the scenario.
    -nn, --no-notify          Disable sending any notifications defined for the
                              scenario.
    --help                    Show this message and exit.
```

```
teflo notify -s data_folder/.results/results.yml -w .
```

Init

Initializes a teflo project under a directory called teflo_workspace, unless the user provides a dir name using the -d/–dirname flag.

Creates the necessary files, includes teflo.cfg, ansible.cfg, ansible playbooks, and some scenario files to do provision, orchestrate and execute jobs.

```
teflo init --help
Usage: teflo init [OPTIONS]

    Initializes a teflo project in your workspace.

Options:
    -d, --dirname             Directory name to create teflo initial files in it. By
                              default, the name is teflo_workspace.
    --help                    Show this message and exit.
```

```
teflo init
```

```
teflo init --dirname new_project
```

After you run *teflo init* command the project file tree will look like this:

```
.
├── execute
│   ├── add_two_numbers.sh
│   ├── README.rst
│   ├── SampleTest.xml
│   ├── scenario.yml
│   └── teflo.cfg
├── orchestrate
│   ├── ansible
│   │   ├── mock_kernel_update.yml
│   │   └── system_info.yml
│   ├── ansible.cfg
│   ├── README.rst
│   ├── scenario.yml
│   └── teflo.cfg
└── provision
    ├── README.rst
```

(continues on next page)

(continued from previous page)

```
    ├── scenario.yml
    └── teflo.cfg
```

You can use the examples using the README.rst files in the same folder.

Alias

Teflo allows the use of alias to run predefined commands(Similar to git) To use it add alias block to the teflo.cfg file:

```
[alias]
dev_run=run -s scenario.yml --log-level debug --iterate-method by_depth
prod_run=show -s scenario.yml --list-labels
```

To run dev_run alias:

```
$ Teflo alias dev_run
```

Getting Started Examples

This section contains examples to help get you started with teflo. A separate examples repository contains all the examples that will be covered below. Please clone this repository into your local environment to use.

Provision

Please visit the following page for complete examples on using provision task.

Orchestrate

Please visit the following page for complete examples on using teflos orchestrate task.

Execute

Please visit the following page for complete examples on using teflos execute task.

Resource_check

Please visit the following page for complete examples on using teflos resource_check option.

2.4 Detailed Information

2.4.1 Scenario Descriptor

This page is intended to explain the input to teflo. The goal and focus behind teflos input is to be simple and transparent. It uses common language to describe the entire scenario (E2E). The input is written in YAML. The term used to reference teflo's input is a scenario descriptor file. You will hear this throughout teflo's documentation.

Every scenario descriptor file is broken down into different sections. Below is a table of the keys that correlate to the different sections.

Key	Description	Type	Required
name	The name of the scenario descriptor file.	Strin	True
description	A description of the intent of the scenario.	Strin	False
resource_che	A list of external resources the scenario depends on that Teflo can check in Semaphore before running the scenario.	List	False
include	A list of scenario descriptor files that should be included when running the scenario.	List	False
provision	A list that contains blocks of Asset definitions that should be dynamically provisioned or statically defined to be used by the rest of the scenario.	List	False
orchestrate	A list that contains blocks of Action definitions that define scripts or playbooks that should be run to configure the assets defined in the provision.	List	False
execute	A list that contains blocks of Execute definitions that define scripts, commands, or playbooks that execute tests on the appropriately configured assets.	List	False
report	A list that contains blocks of Report definitions that should be run to import test artifacts collected during test execution to a desired reporting system.	List	False

Each section relates to a particular component within teflo. You can learn about this at the architecture page. Below are sub pages which go into further detail explaining the different sections.

Resource Check

Teflo's Resource Dependency Check Section is optional. It is run during the Validate task Resource_check is a dictionary which takes in three keys **monitored_services (formerly was service), playbook, script**

Monitored_Services

User can define a list of external components to check if their status is up or not. if all components are up the scenario will be executed .If one or more components are down the scenario will exit with an error. It will also indicate if a component name given is invalid.

The key "resource_check_endpoint" must be set in the teflo.cfg file to actually perform check. If not set this section is ignored. The "resource_check_endpoint" must be the URL of a "Cachet" status page endpoint. Component names must be valid for that status page

```
[defaults]
log_level=info
workspace=.
data_folder=.teflo
resource_check_endpoint=<URL>
```

Playbook/ Script

User can put in a list of customized playbooks or scripts to validate certain things before starting their scenario. if any of the user defined validation playbook/scripts fail the scenario will not be run.

All playbooks and scripts are run only on the localhost from where teflo is being executed. Teflo will not be able to take any output from these scripts/playbooks and make any decisions based on that Teflo will consider the resource _check successfull or not based on the return code received after running the playbook or script

Teflo uses the ansible to run these playbooks and scripts. User should define playbooks and scripts similar to how it is defined in the Execute section of Teflo

Example 1

Using service, playbook, script

```
---
name: Example Discriptor
description: Descriptor file with resource check section

resource_check:
  monitored_services:
    - polarion
    - umb
  playbook:
    - name: ansible/list_block_devices.yml
      ansible_options:
        extra_vars:
          X: 18
          Y: 12
          ch_dir: ./scripts/
    - name: ansible/tests/test_execute_playbook.yml
      ansible_options:
        extra_vars:
          X: 12
          Y: 12
          ch_dir: ../../scripts/
  script:
      - name: ./scripts/hello_world1.py Teflo_user
        executable: python
      - name: ./scripts/add_two_numbers.sh X=15 Y=15


provision:
   .
   .
   .

orchestrate:
   .
   .
   .
```

(continues on next page)

(continued from previous page)

```
execute:
   .
   .
   .
```

Example 2

Using service

```
name: Example Discriptor
description: Descriptor file with resource check section

resource_check:
  monitored_services: ['polarion', 'umb']

provision:
   .
   .
   .

orchestrate:
   .
   .
   .
```

Credentials

For each resource that needs to be provisioned or artifact that needs to be imported, credentials are required. These credentials will be set in the required teflo.cfg file, and the credential name will be referenced in your scenario descriptor file in the provision section for each resource or artifact that is defined.Or you can set the credentials from a separate file

Define credential from a separate file

You can also define the credentials by creating a credential file (For example, credential.keys) and put all the credentials there. Users need to encrypt this credentials file using ansible-vault. The path for this file needs to be provided in the teflo.cfg as **CREDENTIAL_PATH**. The ansible-vault password needs to be provided in the teflo.cfg file as **VAULTPASS**. These values are present under the default section of the teflo.cfg file.

You need to define the **CREDENTIAL_PATH** and **VAULTPASS** fields in the **teflo.cfg**.

Note: For the VAULTPASS, you can also export it to be an enviroment variable, so you can protect the password

the credentials can be either put in teflo.cfg OR put providing a separate credentials file. These are mutually exclusive

Example:

```
[defaults]
log_level=debug
data_folder=teflo_data/
workspace=.
inventory_folder=css_psi_customerzero/
CREDENTIAL_PATH=credentials.key
VAULTPASS=abc
```

Beaker Credentials

For Beaker, the following table is a list of required and optional keys for your credential section in your teflo.cfg file. You must set either keytab and keytab_principal or username and password:

Key	Description	Type	Re-quired
hub_url	The beaker server url.	String	True
keytab	name of the keytab file, which must be placed in the scenario workspace directory.	String	False
keytab_principal	The principal value of the keytab.	String	False
username	Beaker username.	String	False
password	Beaker username's password.	String	False
ca_cert	path to a trusted certificate file	String	False

Below is an example credentials section in the teflo.cfg file. If the credential was defined as below, it should be referenced in your teflo scenario descriptor by the host as **credential: beaker-creds**:

```
[credentials:beaker-creds]
hub_url=<hub_url>
keytab=<keytab>
keytab_principal=<keytab_principal>
username=<username>
password=<password>
ca_cert=<ca_cert_path>
```

The following is an example of a resource in the scenario descriptor file that references this credential:

```
---

name: beaker resource
description: define a teflo host beaker resource to be provisioned

provision:
  - name: beaker-node
    groups: node
    provisioner: beaker-client
    credential: beaker-creds
    arch: x86_64
    distro: RHEL-7.5
    variant: Server
    whiteboard: teflo beaker resource example
```

(continues on next page)

(continued from previous page)

```
    jobgroup: '{{ jobgroup }}'
    username: '{{ username }}'
    password: '{{ password }}'
    host_requires_options:
      - "force={{ host_fqdn }}"
    ksappends:
      - |
        %post
        echo "This is my extra %post script"
        %end
```

OpenStack Credentials

For OpenStack, the following table is a list of required and optional keys for your credential section in your teflo.cfg file.

Key	Description	Type	Required
auth_url	The authentication URL of your OpenStack tenant. (identity)	String	True
tenant_name	The name of your OpenStack tenant.	String	True
username	The username of your OpenStack tenant.	String	True
password	The password of your OpenStack tenant.	String	True
region	The region of your OpenStack tenant to authenticate with.	String	False
domain_name	The name of your OpenStack domain to authenticate with. When not set teflo will use the 'default' domain	String	False
project_id	The id of your OpenStack project.	String	False
project_domain	The id of the project domain.	String	False

```
[credentials:openstack-creds]
auth_url=<auth_url>
tenant_name=<tenant_name>
username=<username>
password=<password>
region=<region>
domain_name=<domain_name>
project_id=<project id>
project_domain_id=<project_domain_id>
```

The following is an example of a resource in the scenario descriptor file that references this credential:

```
      ansible_user: root
      ansible_ssh_private_key_file: "keys/{{ key_name }}"

# openstack scenario
---

name: openstack resource
description: define a teflo host openstack resource to be provisioned

```

(continues on next page)

(continued from previous page)

```
provision:
  - name: openstack-node
    groups: node
    provisioner: openstack-libcloud
    credential: openstack-creds
    image: rhel-7.5-server-x86_64-released
    flavor: m1.small
    networks:
      - '{{ network }}'
    floating_ip_pool: "10.8.240.0"
    keypair: '{{ keypair }}'
```

```
---

name: openstack resource
description: define a teflo host openstack resource to be provisioned

provision:
  - name: openstack-node
    groups: node
    provisioner: openstack-libcloud
    credential: openstack-creds
    image: rhel-7.5-server-x86_64-released
    flavor: m1.small
    networks:
      - '{{ network }}'
    floating_ip_pool: "10.8.240.0"
    keypair: '{{ keypair }}'
    ansible_params:
      ansible_user: cloud-user
      ansible_ssh_private_key_file: "keys/{{ keypair }}"
```

Email Credentials

For email-notifier, the following table is a list of required and optional keys for your credential section in your teflo.cfg file.

Key	Description	Type	Required
smtp_host	The SMTP Server should be used to send emails.	String	True
smtp_port	The port number to use if not using the default port number.	String	False
smtp_user	The username to connect to your SMTP Server if authentication required	String	False
smtp_password	The password of the SMTP user to authenticate if required.	String	False
smtp_starttls	Whether to put the connection in TLS mode.	Boolean	False

```
[credentials:email-creds]
smtp_host=<smtp server fqdn>
smtp_port=<port number>
```

(continues on next page)

(continued from previous page)

```
smtp_user=<user>
smtp_password=<password>
smtp_starttls=<True/False>
```

Including Scenarios

Overview

The __*Include*__ section is introduced to provide a way to include common steps of provisioning, orchestration, execute or reports under one or more common scenario files. This reduces the redundancy of putting the same set of steps in every scenario file. Each scenario file is a single node of the whole __*Scenario Graph*__

When running a scenario that is using the include option, several results files will be generated. One for each of the scenarios. the included scenario will use the scenario's name as a prefix. e.g. common_scenario_results.yml where common_scenario is the name of the included scenario file. All these files will be stored in the same location. This allows users to run common.yml(s) once and their result(s) can be included in other scenario files saving time on test executions. Also see Teflo Output

Note: For any given task the included scenario is checked and executed first followed by the parent scenario. For example, for Orchestrate task, if you have an orchestrate section in both the included and main scenario, then the orchestrate tasks in included scenario will be performed first followed by the orchestrate tasks in the main scenario.

Example 1

You want to separate out provision of a set of resources because this is a common resource used in all of your scenarios.

```
---
name: Include_example1
description: Descriptor file with include section

resource_check:

include:
   - provision.yml

orchestrate:
   .
   .
   .
execute:
   .
   .
   .
report:
   .
   .
   .
```

The provision.yml could look like below

```
---
name: common-provision
description: 'common provisioning of resources used by the rest of the scenarios.'

provision:
- name: ci_test_client_b
  groups:
  - client
  - vnc
  provisioner: beaker-client
```

Example 2

You want to separate out provision and orchestrate because this is common configuration across all your scenarios.

```
---
name: Include_example2
description: Descriptor file with include section

include:
    - provision.yml
    - orchestrate.yml

execute:
   .
   .
   .

report:
   .
   .
   .
```

The orchstrate.yml could look like below

```
# Example 9
---
orchestrate:
  - name: orc_script
    description: creates a local dir
    ansible_options:
      extra_args: -c -e 12
    ansible_script:
      name: scripts/create_dir.sh
    hosts: localhost
    orchestrator: ansible

# Example 10
```

Example 3

You've already provisioned a resource from a scenario that contained just the provision and you want to include it's results.yml in another scenario.

```
---
name: Include_example3
description: Descriptor file with include section

include:
  - /var/lib/workspace/teflo_data/.results/common-provision_results.yml

orchestrate:
  .
  .
  .

execute:
   .
   .
   .

report:
   .
   .
   .
```

The common-provision_results.yml could look like below

```
---
name: common-provision
description: 'common provisioning of resources used by the rest of the scenarios.'

provision:
- name: ci_test_client_a
  description:
  groups:
  - client
  - test_driver
  provisioner: linchpin-wrapper
  provider:
    count: 1
    credential: aws-creds
    name: aws
    region: us-east-2
    hostname: ec2-host.us-east-2.compute.amazonaws.com
    tx_id: 44
    keypair: ci_aws_key_pair
    node_id: i-0f452f3479919d703
    role: aws_ec2
    flavor: t2.nano
    image: ami-0d8f6eb4f641ef691
  ip_address:
```

(continues on next page)

(continued from previous page)

```
    public: 13.59.32.24
    private: 172.31.33.91
  ansible_params:
    ansible_ssh_private_key_file: keys/ci_aws_key_pair.pem
    ansible_user: centos
  metadata: {}
  workspace: /home/dbaez/projects/teflo/e2e-acceptance-tests
  data_folder: /var/lib/workspace/teflo_data/fich6j1ooq
```

Example 4

You want to separate out provision and orchestrate because this is common configuration across all your scenarios but with this particular scenario you want to also a run a non-common orchestration task.

```
---
name: Include_example4
description: Descriptor file with include section

include:
- provision.yml
- orchestrate.yml

orchestrate:
- name: ansible/ssh_connect.yml
    description: "setup key authentication between driver and clients"
    orchestrator: ansible
    hosts: driver
    ansible_options:
      skip_tags:
      - ssh_auth
      extra_vars:
        username: root
        password: redhat
    ansible_galaxy_options:
      role_file: roles.yml

execute:
  .
  .
  .

report:
  .
  .
  .
```

Example 5

You can use jinja templating like below

```
---
name: Include_example5
description: Descriptor file with include section

include:
  - teflo/stack/provision_localhost.yml
  - teflo/stack/provision_libvirt.yml
{% if true %}  - teflo/stack/orchestrate-123.yml{% endif %}
  - teflo/stack/orchestrate.yml

orchestrate:
- name: ansible/ssh_connect.yml
    description: "setup key authentication between driver and clients"
    orchestrator: ansible
    hosts: driver
    ansible_options:
      skip_tags:
      - ssh_auth
      extra_vars:
        username: root
        password: redhat
    ansible_galaxy_options:
      role_file: roles.yml

execute:
  .
  .
  .

report:
  .
  .
  .
```

Scenario Graph Explanation

There are two ways of executing teflo scenarios, which are __by_level and by__depth. User can modify how the scenarios are executed by changing the setting __included_sdf_iterate_method__ in the teflo.cfg , as shown below, by_level is set by default if you don't specify this parameter

```
[defaults]
log_level=info
workspace=.
included_sdf_iterate_method = by_depth
```

All blocks(provision, orchestrate, execute, report) in a senario descriptor file will be executed together for each scenario, in case there are included scenarios

Note: Scenarios should be designed such that the dependent(which you want it to run first) scenario should be at the child level. In the below example if sdf13 has the provisioning information and the orchestrate block which uses these provisioned assets can be in scenario which is at a higher level, but not the other way round

Example

```
                                sdf

          /                      |                     \

        sdf1                    sdf7                     sdf

  /       |       \             /   \                 /    |    \

sdf3    sdf8      sdf5       sdf10 sdf11          sdf4   sdf9  sdf6

        /    \

    sdf12 sdf13
```

The above is an complex include usage. Consider sdf1-sdf13 are different included scenarios and sdf is the main scenario

by_level

The execution order will be 12,13,3,8,5,10,11,4,9,6,1,7,2,0

by_depth

The execution order will be 12,13,3,8,5,1,10,11,7,4,9,6,2,0

Remote Include

You can include teflo workspace from remote server(currently only support for git)

Example SDF

```
---
name: remote_include_example
description: include remote sdf from git server

remote_workspace:
  - workspace_url: git@github.com:dno-github/remote-teflo-lib1.git
    alias_name: remote
    # the alias_name should not be the same as local folder, it will collide
```

(continues on next page)

(continued from previous page)

```
  - workspace_url: https://github.com/dno-github/remote-teflo-lib1.git
    alias_name: remote2

include:
  - "remote/sdf_remote.yml"

name: sdf using remote include
description: "Provision step"

provision:
  - name: from_local_parent
    groups: localhost
    ip_address: 127.0.0.1
    ansible_params:
      ansible_connection: local


execute:
  .
  .
  .

report:
  .
  .
```

Note: When using ssh to clone (example above “remote”), user need to use GIT_SSH_COMMAND= in teflo.cfg.

Example teflo.cfg

```
[defaults]
log_level=debug
workspace=.
data_folder=.teflo
# This is the directory for all downloaded remote workspaces
remote_workspace_download_location=remote_dir
# if you set this to False, the downloaded remote workspace
# will not be removed after the teflo job is done. And teflo
# will automatically reuse the downloaded workspace if you run
# the same job again(skip the download process, could potentially
# save your time)
# when using SSH instead of HTTPS to clone remote workspace, private key path can be set␣
↪from here.
GIT_SSH_COMMAND=/home/user/keys/private_k
CLEAN_CACHED_WORKSPACE_AFTER_EACH_RUN = False
```

workspace_url is the url of the git repo(your teflo workspace), alias_name is the name which you want to use in include section .. note:

```
The alias_name should not be the same as local folder, it will collide
```

Provision

Overview

The input for provisioning will depend upon the type of resource you are trying to provision. The current support for provisioning resources are: *Beaker* and *OpenStack*. Resources can also be provisioned using the *Linchpin* provisioner.

Provision Resource

Each resource defined within a provision section can have the following common keys, and the table below will describe whether the keys are required or optional:

```
---
provision:
  - name: <name>
    groups: <groups>
    provisioner: <provisioner>
    metadata: <dict_key_values>
    ansible_params: <dict_key_values>
```

Key	Description	Type	Re-quired
name	The name of the asset to provision.	String	True
groups	The names of the groups for the asset. Used to assign host assets to groups when generating the Ansible inventory files.	List	False
provi-sioner	The name of the provisioner to use to provision assets.	String	False
metadata	Data that the resource may need access to after provisioning is finished. This data is passed through and is not modified by teflo framework.	Dict	False
ansi-ble_params	Ansible parameters to be used within a inventory file to control how ansible communi-cates with the host asset.	Dict	False

Provisioner

This key is will be a requirement to specify the name of provisioner plugin being used

Provider

Attention: The provider key is no longer a requirement for all provisioners.

Groups

Teflo groups are the equivalent of Ansible groups.

Originally this key was named as *role*. the schema was changed from role to groups to better reflect what the purpose of this parameter is intended for.

Groups is not a requirement for all asset types. This should only be specified for host assets like VMs or Baremetal Systems that have an ip and will be acted on later on during Orchestrate or Execute. Assets like networks, storage, security key, etc. do not and should not be assigned a groups to avoid polluting the Ansible inventory file with empty groups.

You can associate a number of groups to a host in a couple of different ways. First is to define your groups in a comma separated string

```
---
provision:
- name: ci_test_client_b
  groups: client, vnc
  provisioner: beaker-client
```

Here we have defined a list of groups.

```
---
provision:
- name: ci_test_client_b
  groups:
  - client
  - vnc
  provisioner: beaker-client
```

Provisioning Systems from Beaker

Credentials

To authenticate with Beaker, you will need to have your Beaker credentials in your teflo.cfg file, see Beaker Credentials for more details.

Beaker Resource

The following shows all the possible keys for defining a provisioning resource for Beaker using the **beaker-client** provisioner:

```
---
provision:
  - name: <name>
    groups: <groups>
    provisioner: beaker-client
    credential: <credential>
    arch: <arch>
    variant: <variant>
    family: <family>
    distro: <os_distro>
    whiteboard: <whiteboard>
    jobgroup: <group_id>
    tag: <tag>
    host_requires_options: [<list of host options>]
    key_values: [<list of key/value pairs defining the host>]
    distro_requires_options: [<list of distro options>]
    virtual_machine: <True or False>
    virt_capable: <True or False>
    priority: <priority of the job>
    retention_tag: <retention tag>
    timeout: <timeout val for Beaker job>
    kernel_options: [<list of kernel options>]
    kernel_post_options: [<list of kernel post options>]
    kickstart: < Filename of kickstart file>
    ignore_panic: <True or False>
    taskparam: [<list of task parameter settings>]
    ksmeta: [<list of kick start meta OPTIONS>]
    ksappends: [<list of kickstart append scripts>]
    metadata: <dict_key_values>
    ansible_params: <dict_key_values>
```

Key	Description	Type	Re-quired
credential	The name of the credentials to use to boot node. This is the one defined in the credentials section of the teflo config file.	String	True
arch	The arch of the node.	String	True
variant	The OS variant of the node.	String	True
family	The OS family of the node. (family or distro needs to be set)	String	False
distro	The specific OS distribution. (family or distro needs to be set)	String	False
whiteboard	The name to set for the Beaker whiteboard to help identify your job.	String	False
jobgroup	The name of the beaker group to set, of who can see the machines and used for machine searching.	String	False
tag	The name of a tag to get the correct OS (i.e. RTT-ACCEPTED).	String	False
host_requires_op	List of host options with the format:[“<key><operand><value>”].	List	False
key_values	List of key/value pairs defining the host, with the format:[“<key><operand><value>”].	List	False
dis-tro_requires_opti	List of OS options with the format:[“<key><operand><value>”].	List	False
kernel_options	List of Beaker kernel options during install with the format:[“<key><operand><value>”]	List	False
ker-nel_options_post	List of Beaker kernel options after install with the format:[“<key><operand><value>”]	List	False
vir-tual_machine	Look for a node that is a virtural machine.	Boolea	False
virt_capable	Look for a machine that is virt capable.	Boolea	False
priority	Set the priority of the Beaker job.	String	False
retention_tag	Set the tag value of how long to keep the job results.	String	False
ssh_key	Name of the ssh key to inject to the test system, file must be placed in your scenario workspace directory.	String	False
username	username of the Beaker machine, required if using **ssh_key**.	String	False
password	password of the Beaker machine, required if using **ssh_key**.	String	False
timeout	Set a value of how long to wait for the Beaker job in seconds.(Default is 8hrs = 28800)	Boolea	False
kickstart	Name of the kickstart template for installation, the file must be placed in your scenario workspace directory.	String	False
ignore_panic	Do not abort job if panic message appears on serial console	Boolea	False
taskparam	parameter settings of form NAME=VALUE that will be set for every task in job	List	False
ksmeta	kickstart metadata OPTIONS for when generating kickstart	List	False
ksappends	partial kickstart scripts to append to the main kickstart file	List	False

Example

```
---

name: beaker resource
description: define a teflo host beaker resource to be provisioned

provision:
  - name: beaker-node
    groups: node
    provisioner: beaker-client
```

(continues on next page)

(continued from previous page)

```
    credential: beaker-creds
    arch: x86_64
    distro: RHEL-7.5
    variant: Server
    whiteboard: teflo beaker resource example
    jobgroup: '{{ jobgroup }}'
    username: '{{ username }}'
    password: '{{ password }}'
    host_requires_options:
      - "force={{ host_fqdn }}"
    ksappends:
      - |
        %post
        echo "This is my extra %post script"
        %end
    ssh_key: "keys/{{ key_name }}"
    ansible_params:
      ansible_user: root
      ansible_ssh_private_key_file: "keys/{{ key_name }}"
```

Provisioning Systems from OpenStack

Credentials

To authenticate with OpenStack, you will need to have your OpenStack credentials in your teflo.cfg file, see OpenStack Credentials for more details.

OpenStack Resource

The following shows all the possible keys for defining a provisioning resource for OpenStack using the **openstack-libcloud** provisioner:

```
---
provision:
  - name: <name>
    groups: <groups>
    provisioner: openstack-libcloud
    metadata: <dict_key_values>
    ansible_params: <dict_key_values>
    credential: openstack-creds
    image: <image>
    flavor: <flavor>
    networks: <networks>
    floating_ip_pool: <floating_ip_pool>
    keypair: <keypair>
    server_metadata: <dict_key_values>
```

Key	Description	Type	Required
credential	The name of the credentials to use to boot node. This is the one defined in the credentials section of the teflo config file.	String	True
image	The name or ID of the image to boot.	String	True
flavor	The name or ID of the flavor to boot.	String	True
networks	The name of the internal network to attach node too.	List	True
floating_ip_pool	The name of the external network to attach node too.	String	False
keypair	The name of the keypair to associate the node with.	String	True
server_metac	Metadata to associate with the node.	Dict	False

Example

```
---

name: openstack resource
description: define a teflo host openstack resource to be provisioned

provision:
  - name: openstack-node
    groups: node
    provisioner: openstack-libcloud
    credential: openstack-creds
    image: rhel-7.5-server-x86_64-released
    flavor: m1.small
    networks:
      - '{{ network }}'
    floating_ip_pool: "10.8.240.0"
    keypair: '{{ keypair }}'
    server_metadata:
      provisioned_by: "teflo"
      build_url: "jenkins.com/build/123"
    ansible_params:
      ansible_user: cloud-user
      ansible_ssh_private_key_file: "keys/{{ keypair }}"
```

Provisioning Openstack Assets using teflo_openstack_client_plugin

Teflo is offers a plugin **teflo_openstack_client_plugin**. to provision openstack resources. This plugin utilizes the openstackclient to provision resources.

User can now install this plugin from Teflo

```
$ pip install teflo[openstack-client-plugin]
```

In your scenario descriptor file specify the **provisioner** key in your provision section.

```
provisioner: openstack-client
```

For more information on how to install plugin and setup the scenario descriptor file for using this plugin, please refer here <here

Provisioning Assets with Linchpin

Users can provision assets using all Linchpin supported providers using the Linchpin plugin.

First the plugin must be installed.

User can now install this plugin from Teflo

```
$ pip install teflo[linchpin-wrapper]
```

You can also refer to the plugin documentation directly

In your scenario file specify the **provisioner** key in your provision section.

```
provisioner: linchpin-wrapper
```

Specify any of the keys supported by the linchpin provisioner.

Note: **provider** key is no longer supported to be used with linchpin provisoiner It is highly recommended that users migrate to using the new set of linchpin provisioner keys.

Note: Due to Linchpin's lack of transactional concurrency support in their database it is recommended to provision resources sequentially. Refer to the task_concurrency setting in the teflo.cfg to switch the provision task execution to be sequential.

Credentials

Since Linchpin support multiple providers, each provider supports different types of parameters. Linchpin also comes with it's own ability to pass in credentials. To be flexible, we support the following options

- You can define the credentials in the *teflo.cfg* and reference them using the Teflo *credential* key. In most cases, Teflo will export the provider specific credential environmental variables supported by Linchpin/Ansible.
- You can use the Linchpin *credentials* option and create the credentials file per Linchpin provider specification.
- You can specify no *credential* or *credentials* key and export the specific provider credential environmental variables supported by Linchpin/Ansible yourself.

For more information refer to the plugins credential document section.

Examples

Below we will just touch on a couple examples. You can see the rest of the examples in the plugin documentation.

Example 1

This example uses a PinFile that has already been developed with specific targets in the pinfile.

```
---
provision:
- name: db2_dummy
  provisioner: linchpin-wrapper
  pinfile:
    path: openstack-simple/PinFile
    targets:
      - openstack-stage
      - openstack-dev
```

Example 2

```
---
provision:
- name: db2_dummy
  provisioner: linchpin-wrapper
  credential: osp-creds
  groups:
    - example
  resource_group_type: openstack
  resource_definitions:
    - name: {{ instance | default('database') }}
      role: os_server
      flavor: {{ flavor | default('m1.small') }}
      image:  rhel-7.5-server-x86_64-released
      count: 1
      keypair: {{ keypair | default('db2-test') }}
      networks:
        - {{ networks | default('provider_net_ipv6_only') }}
  ansible_params:
    ansible_user: cloud-user
    ansible_ssh_private_key_file: keys/{{ OS_KEYPAIR }}
```

Using Linchpin Count

Teflo supports Linchpin's count feature to create multiple resources in a single Asset block. Refer to the example.

Example

To create 2 resources in openstack **count: 2** is added. It's important to note that when multiple resources are generated Teflo will save them as two distinct assets. Assets created using count will be suffixed with a digit starting at 0 up to the number of resources.

This example will provision 2 resources *openstack-node_0* and *openstack-node_1*

By default count value is 1.

```yaml
provision:
- name: openstack-node
  groups: node
  provisioner: linchpin-wrapper
  resource_group_type: openstack
  resource_definitions:
     - name: openstack
       credential: openstack-creds
       image: rhel-7.5-server-x86_64-released
       flavor: m1.small
       networks:
        - '{{ network }}'
       count: 2
```

the output of results.yml

```yaml
provision:
- name: openstack-node_0
  groups: node
  provisioner: linchpin-wrapper
  resource_group_type: openstack
  resource_definitions:
     - name: openstack
       credential: openstack-creds
       image: rhel-7.5-server-x86_64-released
       flavor: m1.small
       networks:
        - '{{ network }}'
       count: 2

- name: openstack-node_1
  groups: node
  provisioner: linchpin-wrapper
  resource_group_type: openstack
  resource_definitions:
     - name: openstack
       credential: openstack-creds
       image: rhel-7.5-server-x86_64-released
       flavor: m1.small
```

(continues on next page)

(continued from previous page)

```
        networks:
         - '{{ network }}'
        count: 2
```

Generating Ansible Inventory

Both Teflo and Linchpin have the capability to generate inventory files post provisioning. For those that want Teflo to continue to generate the Inventory file and use the Linchpin provisioner as just a pure provisioner can do so by specifying the following keys

- groups
- ansible_params

Refer to Example 2 above in the examples section.

For those that want to use Linchpin to generate the inventory file. You must do the following

- Specify the **layout** key and either provide a dictionary of a Linchpin layout or provide a path to a layout file in your workspace.
- Do NOT specify **groups** and **ansible_params** keys

Refer to example 6 and example 8 in the Linchpin plugin documents to see the two variations.

Defining Static Machines

There may be scenarios where you already have machines provisioned and would like teflo to use these static machines. This option is supported in teflo. The main key that needs to be stated is the **ip_address**.

The following is an example of a statically defined machine:

Example

```
---
name: static resource
description: define a static resource to be used throughout teflo

provision:
  - name: static-node
    groups: node
    ip_address: 1.1.1.1
    ansible_params:
      ansible_user: root
      ansible_ssh_private_key_file: "keys/{{ key_name }}"
```

There may also be a scenario where you want to run cmds or scripts on the local system instead of the provisioned resources. Refer to the localhost page for more details.

Orchestrate

Teflo's orchestrate section declares the configuration to be be performed in order to test the systems properly.

First lets go over the basic structure that defines a configuration task.

```
---
orchestrate:
  - name:
    description:
    orchestrator:
    hosts:
```

The above code snippet is the minimal structure that is required to create a orchestrate task within teflo. This task is translated into a teflo action object which is part of the teflo compound. You can learn more about this at the architecture page. Please see the table below to understand the key/values defined.

Key	Description	Type	Required	Default
name	The name of the action you want teflo to execute	String	Yes	n/a
description	A description of what the resource is trying to accomplish	String	No	n/a
orchestrator	The orchestrator to use to execute the action (name) you defined above	String	No (best practice to define this!)	ansible
hosts	The list of hosts where teflo will execute the action against	List	Yes	n/a
environment_vars	Additional environment variables to be passed during the orchestrate task	dict	No	environment variables set prior to starting the teflo run are available

Hosts

You can associate hosts to a given orchestrate task a couple of different ways. First is to define your hosts in a comma separated string.

```
---
orchestrate:
  - name: register_task
    hosts: host01, host02
    ansible_playbook:
      name: rhsm_register.yml
```

You can also define your hosts as a list.

```
---
orchestrate:
  - name: -_taskregister
    hosts:
      - host01
      - host02
    ansible_playbook:
      name: rhsm_register.yml
```

It can become tedious if an orchestrate task needs to be performed on multiple or all hosts within the scenario and you have many hosts declared. Teflo provides you with the ability to run against a group of hosts or all hosts. To run against multiple hosts use the name defined in the **groups** key for your hosts or use **all** to run against all hosts. This eliminates the need to define every host per multiple tasks. It can be either in string or list format.

```
---
orchestrate:
  - name: register_task
    hosts: all
    ansible_playbook:
      name: rhsm_register.yml
```

```
---
orchestrate:
  - name: task1
    hosts: clients
    ansible_playbook:
      name: rhsm_register.yml
```

Re-running Tasks and Status Code

You may notice in your results.yml that each orchestrate block has a new parameter

```
status: 0
```

When teflo runs any of the defined orchestration tasks successfully a status code of 0 will be set. If an orchestration task fails, teflo will set the status to 1. The next time you re-run the Orchestration task teflo will check for any orchestration tasks with a failed status and start the orchestration process from there.

This is useful when you have a long configuration process and you don't want to start over all the way from the beginning. If at some point you would rather have the orchestration process start from the beginning, you can modify the status code back 0.

Since teflos development model is plug and play. This means different orchestrator's could be used to execute configuration tasks declared. Ansible is Teflo's default orchestrator. Its information can be found below.

Ansible

Ansible is teflos default orchestrator. As we mentioned above each task has a given **name** (action). This name is the orchestrate task name.

Teflo uses these keywords to detect to ansible playbook, script or shell command **ansible_playbook, ansible_script, ansible_shell** respectively. Please refer *here* to get an idea on how to use the keys

In addition to the required orchestrate base keys, there are more you can define based on your selected orchestrator.Lets dive into them..

Key	Description	Type	Required	Default
ansible_option	Additional options to provide to the ansible orchestrator regarding the task (playbook) to be executed	Dictionary	No	n/a
ansible_galaxy	Additional options to provide to the ansible orchestrator regarding ansible roles and collections	Dictionary	No	n/a
ansible_script	scribt to be executed	Dictionary	(Not required; however, one of the following must be defined: ansible_shell/ansible_script/ansible_playbook)	False
ansible_playbc	playbook to be run.	dictionary	(Not required; however, one of the following must be defined: ansible_shell/ansible_script/ansible_playbook)	False
ansible_shell	shell commands to be run.	list of dictionary	(Not required; however, one of the following must be defined: ansible_shell/ansible_script/ansible_playbook)	False

The table above describes additional key:values you can set within your orchestrate task. Each of those keys can accept additional key:values.

Use Ansible group_vars

Ansible can set variables to each host with different ways, one of them is using the group_vars file.

```
---

ansible_user: fedora
```

Note: For more information read from Ansible Docs.

Teflo will look for group_vars dir inside workspace/ansible:

workspace/ansible/group_vars/example

Use Playbook Within A Collection

We can Use playbook within a collection with Fully Qualified Collections Name. When running a playbook using fqcn, Teflo will first check if the collection exist and will try to download it if needed.

Example

Lets first call the playbook in our orchestrate task:

```
  # use FQCN and collection install
 - name: Example 1                              # action name
   description: "use fqcn"                      # describes what is being performed on␣
↪the hosts
   orchestrator: ansible                        # orchestrator module to use in this␣
↪case ansible
   hosts:                                       # hosts which the action is executed on
     - all                                      # ref above ^^ to all hosts : provision.*
   ansible_playbook:
     name: namespace.collection1.playbook1        # playbook name(Using FQCN)
   ansible_galaxy_options:
     role_file: requirements.yml                # A .yml file to describe␣
↪collection(name,type,version)
```

the requirements.yml should look like:

```
---
collections:
  - name: https://github.com/collection/path
    type: git
    version: main
```

Note: For more information read from Ansible Docs.

By default Teflo will install collections under "workspace/collections/" To change default use the ansible.cfg file:

collections_paths = ./wanted_coll_path

Teflo Ansible Configuration

In the teflo configuration file, you can set some options related to ansible. These values should be set in the **[orchestrator:ansible]** section of the teflo.cfg file. The following are the settings.

Key	Description	Default
log_r	configuration option to delete the ansible log file after configuration is complete. Either way the ansible log will be moved to the user's output directory.	By default this is set to true to delete it.
verbosit	configuration option to set the verbosity of ansible.	Teflo sets the ansible verbosity to the value provided by this option. If this is not set then, teflo will set the verbosity based on teflo's logging level. If logging level is 'info' (default) ansible verbosity is set to **None** else if logging level is 'debug' then ansible verbosity is 'vvvv'.

Note: Teflo can consume the Ansible verbosity level using Ansible's built-in environment variable ANSIBLE_VERBOSITY in addition to consuming it from being defined within teflo.cfg file. If the verbosity value is incorrect within teflo.cfg, teflo will default to the verbosity based on teflo's logging level.

Ansible Configuration

It is highly recommended that every scenario that uses Ansible provide their own ansible.cfg file. This can be used for specific connection requirements, logging, and other settings for the scenario. The following is an example of a configuration file that can be used as a base.

```
[defaults]
# disable strict SSH key host checking
host_key_checking = False

# filter out logs that are not ansible related
log_filter = paramiko,pykwalify,teflo,blaster,urllib3

# set the path to set ansible logs
log_path = ./ansible.log

# set specific privelege escalation if necessary for the scenario
[privilege_escalation]
become=True
become_method=sudo
become_user=test
```

To see all of the settings that can be set see Ansible Configuation Settings.

Ansible Logs

To get ansible logs, you must set the **log_path** in the ansible.cfg, and it is recommended to set the **log_filter** in the ansible.cfg as described to filter out non ansible logs. If you do not set the log path or don't provide an ansible.cfg, you will not get any ansible logs. The ansible log will be added to the **ansible_orchestrate** folder under the logs folder of teflo's output, please see Teflo Output for more details.

Using ansible_script

Orchestrate task uses ansible playbook module to run the user provided scripts. The script name can be given within the **name** key of the ansible_script list of dictionary.

The script parameters can be provided along with name of the script by separating it using space.

Note: The script parameters can also be passed using ansible_options key. But this will be deprecated in the future releases *Example 15*

Extra_args for the script can be provided as a part of the ansible_script list of dictionary or under ansible_options. Please see *Extra_args Example 13 Example 14*

Using ansible_shell

Orchestrate task uses ansible shell module to run the user provided shell commands. ansible_shell takes in a list of dictionaries for the different commands to be run. The shell command can be provided under the *command* key the ansible_shell list of dictionary. Extra_args for the shell command can be provided as a part of the ansible_shell list of dictionary or under ansible_options. Please see *Extra_args Example 12*

When building your shell commands it is important to take into consideration that there are multiple layers the command is being passed through before being executed. The two main things to pay attention to are YAML syntax/escaping and Shell escaping.

When writing the command in the scenario descriptor file it needs to be written in a way that both Teflo and Ansible can parse the YAML properly. From a Teflo perspective it is when the the scenario descriptor is first loaded. From an Ansible perspective its when we pass it the playbook we create, cbn_execute_shell.yml, through to the ansible-playbook CLI.

Then there could be further escapes required to preserve the test command so it can be interpreted by the shell properly. From a Teflo perspective that is when we pass the test command to the ansible-playbook CLI on the local shell using the -e "xcmd='<test_command>'" parameter. From the Ansible perspective its when the shell module executes the actual test command using the shell on the designated system.

Let's go into a couple examples

```yaml
ansible_shell:
  - command: glusto --pytest='-v tests/test_sample.py --junitxml=/tmp/SampleTest.xml'
             --log /tmp/glusto_sample.log
```

On the surface the above command will pass YAML syntax parsing but will fail when actually executing the command on the shell. That is because the command is not preserved properly on the shell when it comes to the *–pytest* optioned being passed in. In order to get this to work you could escape this in one of two ways so that the *–pytest* optioned is preserved.

```
ansible_shell:
  - command: glusto --pytest=\\\"-v tests/test_sample.py --junitxml=/tmp/SampleTest.xml\\
↪\"
            --log /tmp/glusto_sample.log


ansible_shell:
  - command: glusto \\\"--pytest=-v tests/test_sample.py --junitxml=/tmp/SampleTest.xml\\
↪\"
            --log /tmp/glusto_sample.log
```

Here is a more complex example

```
ansible_shell:
    - command: if [ `echo \$PRE_GA | tr [:upper:] [:lower:]` == 'true' ];
               then sed -i 's/pre_ga:.*/pre_ga: true/' ansible/test_playbook.yml; fi
```

By default this will fail to be parsed by YAML as improper syntax. The rule of thumb is if your unquoted YAML string has any of the following special characters :-{}[]!#|>&%@ the best practice is to quote the string. You have the option to either use single quote or double quotes. There are pros and cons to which quoting method to use. There are online resources that go further into this topic.

Once the string is quoted, you now need to make sure the command is preserved properly on the shell. Below are a couple of examples of how you could achieve this using either a single quoted or double quoted YAML string

```
ansible_shell:
    - command: 'if [ \`echo \$PRE_GA | tr [:upper:] [:lower:]\` == ''true'' ];
                then sed -i \"s/pre_ga:.*/pre_ga: true/\" ansible/test_playbook.yml; fi'


ansible_shell:
    - command: "if [ \\`echo \\$PRE_GA | tr [:upper:] [:lower:]\\` == \\'true\\' ];
                then sed \\'s/pre_ga:.*/pre_ga: true/\\' ansible/test_playbook.yml; fi"
```

Note: It is NOT recommended to output verbose logging to standard output for long running tests as there could be issues with teflo parsing the output

Using ansible_playbook

Using the ansible_playbook parameter you can provide the playbook to be run The name of the playbook can be provided as the **name** under the ansible_playbook list of dictionary

Example2 Example12

Note: Unlike the shell or script parameter the test playbook executes locally from where teflo is running. Which means the test playbook must be in the workspace.

Note: Only one action type, either ansible_playbook or ansible_script or ansible_shell is supported per orchestrate task

Extra_args for script and shell

Teflo supports the following parameters used by ansible script and shell modules

Parameters
chdir
creates
decrypt
executable
removes
warn
stdin
stdin_add_newline

Please look here for more info

Ansible Script Module

Ansible Shell Module

vault-password-file

The vault-password-file can be passed using **vault-password-file** under **ansible_options**

```yaml
---
orchestrate:
  - name: rhsm_register.yml
    description: "register systems under test against rhsm"
    orchestrator: ansible
    hosts: all
    ansible_options:
      vault-password-file:
        - "./vaultpass"
```

Extra_vars

Extra variables needed by ansible playbooks can be passed using **extra_vars** key under the **ansible_options** section

```yaml
---
orchestrate:
  - name: rhsm_register.yml
    description: "register systems under test against rhsm"
    orchestrator: ansible
    hosts: all
    ansible_options:
      extra_vars:
        username: kingbob
        password: minions
        server_hostname: server01.example.com
        auto_attach: true
```

Use the **file** key to pass a variable file to the playbook. This file needs to present in teflo's workspace **file** key can be a single file give as string or a list of variable files present in teflo's workspace

```
---
orchestrate:
  - name: rhsm_register.yml
    description: "register systems under test against rhsm"
    orchestrator: ansible
    hosts: all
    ansible_options:
      extra_vars:
        file: variable_file.yml
```

```
---
orchestrate:
  - name: rhsm_register.yml
    description: "register systems under test against rhsm"
    orchestrator: ansible
    hosts: all
    ansible_options:
      extra_vars:
        file:
        - variable_file.yml
        - variable_1_file.yml
```

Note: Teflo can make the variable files declared in the default locations below, to be passed as extra_vars to the ansible playbook in the orchestrate and execute stage

1. defaults section of teflo.cfg
2. var_file.yml under the teflo workspace
3. yml files under the directory vars under teflo workspace

This can be done by setting the following property to True in the defaults section of the teflo.cfg

```
[defaults]
ansible_extra_vars_files=true
```

Example:

Here the default variable file my_default_variable_file.yml is made available as a variable file to be passed as extra_vars to the ansible playbooks being run in the execute and orchestrate stages. If variable file(s) are already being passed to the ansible playbook as a part of ansible_options, this setting will append the default variable files to that list. In the below example for orchestrate stage the file my_default_variable_file.yml is passed along with variable.yml as extra_vars

```
[defaults]
var_file=./my_default_variable_file.yml
ansible_extra_vars_files=true
```

```
orchestrate:
  - name: playbook_2
    description: "run orchestrate step using file key as extra_vars"
    orchestrator: ansible
```

(continues on next page)

(continued from previous page)

```
    hosts: localhost
    ansible_playbook:
      name: ansible/var_test1.yml
    ansible_options:
      extra_vars:
        file: variable.yml

execute:
  - name: playbook_3
    description: "run orchestrate step using file key as extra_vars"
    executor: runner
    hosts: localhost
    playbook:
    - name: ansible/template_host_test_playbook_tasks.yml
```

Ansible Galaxy

Before teflo initiates the *ansible-playbook* command, it will attempt to download any roles or collections based on what is configured within the *ansible_galaxy_options* for the given task. Teflo downloads these dependencies using the *ansible-galaxy* command. In the event the command fails for any reason, teflo will retry the download. A maximum of *2* attempts will be made with a *30* second delay between attempts. Teflo will stop immediately when its unable to download the roles. Reducing potential playbook failures at a later point.

Examples

Lets dive into a couple different examples.

Example 1

You have a playbook which needs to run against x number of hosts and does not require any additional extra variables.

```
---
orchestrate:
  - name: register_task
    description: "register systems under test against rhsm"
    orchestrator: ansible
    ansible_playbook:
      name: rhsm_register.yml
    hosts:
      - host01
      - host02
```

Example 2

You have a playbook which needs to run against x number of hosts and requires additional extra variables.

```
---
orchestrate:
  - name: register_task
    description: "register systems under test against rhsm"
    orchestrator: ansible
    hosts:
      - host01
      - host02
    ansible_playbook:
      name: rhsm_register.yml
    ansible_options:
      extra_vars:
        username: kingbob
        password: minions
        server_hostname: server01.example.com
        auto_attach: true
```

Example 3

You have a playbook which needs to run against x number of hosts and requires only tasks with a tag set to prod.

```
---
orchestrate:
  - name: custom
    description: "running a custom playbook, only running tasks when tag=prod"
    orchestrator: ansible
    hosts:
      - host01
    ansible_playbook:
      name: custom.yml
    ansible_options:
      tags:
        - prod
```

Example 4

You have a playbook which needs to run against x number of hosts and requires only tasks with a tag set to prod and requires connection settings that conflicts with your ansible.cfg.

```
---
orchestrate:
  - name: custom2
    description: "custom playbook, w/ different connection options"
    orchestrator: ansible
    hosts:
      - host07
```

(continues on next page)

(continued from previous page)

```
    ansible_playbook:
      name: custom2.yml
    ansible_options:
      forks: 2
      become: True
      become_method: sudo
      become_user: test_user2
      remote_user: test_user
      connection: paramiko
      tags:
        - prod
```

Example 5

You have a playbook which needs to run against x number of hosts and requires an ansible role to be downloaded.

Note: Although the option is called *role_file:* but it relates both, roles and collections.

```
---
orchestrate:
  - name: register_task
    description: "register systems under test against rhsm"
    orchestrator: ansible
    ansible_playbook:
      name: rhsm_register.yml
    hosts:
      - host01
      - host02
    ansible_galaxy_options:
      role_file: requirements.yml
```

Content of requirements.yml as a dictionary, suitable for both roles and collections:

```
---
roles:
- src: oasis-roles.rhsm

collections:
- name: geerlingguy.php_roles
- geerlingguy.k8s
```

As you can see we defined the role_file key. This defines the ansible requirements filename. Teflo will consume that file and download all the roles and collections defined within.

Note: We can define roles in the req file as a list or as dictionary, Teflo support both ways. but if we chose to set roles as list then we can’t set collections in the same file.

Content of requirements.yml file as a list, only suitable for roles:

```
---
- src: oasis-roles.rhsm
- src: https://gitlab.cee.redhat.com/PIT/roles/junit-install.git
  scm: git
```

An alternative to using the requirements file is you can directly define them using the roles or collections key.

```
---
orchestrate:
  - name: register_task
    description: "register systems under test against rhsm"
    orchestrator: ansible
    hosts:
      - host01
      - host02
    ansible_playbook:
      name: rhsm_register.yml
    ansible_galaxy_options:
      roles:
        - oasis-roles.rhsm
        - git+https://gitlab.cee.redhat.com/oasis-roles/coreos_infra.git,master,oasis_
↪roles.coreos_infra
      collections:
        - geerlingguy.php_roles
        - geerlingguy.k8s
```

It is possible to define both role_file and direct definitions. Teflo will install the roles and collections first from the role_file and then the roles and collections defined using the keys. It is up to the scenario to ensure no problems may occur if both are defined.

Note: If your scenario directory has roles and collections already defined, you do not need to define them. This is only if you want teflo to download roles or collections from sites such as ansible galaxy, external web servers, etc.

Example 6

You have a playbook which needs to run against x number of hosts, requires ansible roles to be downloaded and requires additional extra variables.

```
---
orchestrate:
  - name: register_task
    description: "register systems under test against rhsm"
    orchestrator: ansible
    hosts:
      - host01
      - host02
    ansible_playbook:
      name: rhsm_register.yml
    ansible_options:
      extra_vars:
```

(continues on next page)

(continued from previous page)

```
        username: kingbob
        password: minions
        server_hostname: server01.example.com
        auto_attach: true
    ansible_galaxy_options:
      role_file: roles.yml
```

Attention: Every scenario processed by teflo should define an ansible configuration file. This provides the scenario with the flexibility to easily control portions of ansible.

If you are using the ability to download roles or collections by teflo, you need to set the *roles_path* or the *collections_paths* within your ansible.cfg. If this is not set, the default collections path "<your workspace>/collections/" will be selected.

Here is an example ansible.cfg setting the roles_path and collections_paths to a relative path within the scenario directory.

```
[defaults]
host_key_checking = False
retry_files_enabled = False
roles_path = ./roles
collections_paths = ./collections
```

Example 7

You have a playbook which needs to run against x number of hosts. Prior to deleting the configured hosts. You want to run a playbook to do some post tasks.

```
---
orchestrate:
  - name: register_task
    description: "register systems under test against rhsm"
    orchestrator: ansible
    hosts: all
    ansible_playbook:
      name: rhsm_register.yml
    ansible_options:
      extra_vars:
        username: kingbob
        password: minions
        server_hostname: server01.example.com
        auto_attach: true
    ansible_galaxy_options:
      role_file: roles.yml
    cleanup:
      name: unregister_task
      description: "unregister systems under tests from rhsm"
      orchestrator: ansible
      hosts: all
      ansible_playbook:
```

(continues on next page)

(continued from previous page)

```
        name: rhsm_unregister.yml
      ansible_galaxy_options:
        role_file: roles.yml
```

Example 8

The following is an example of running a script. The following is an example of a script running on the localhost. For localhost usage refer to the`localhost <../localhost.html>`_ page.

```
---
orchestrate:
  - name: orc_script
    description: create a local dir
    ansible_script:
      name: scripts/create_dir.sh
    hosts: localhost
    orchestrator: ansible
```

Example 9

The following builds on the previous example, by showing how a user can add options to the script they are executing (In the example below, the script is run with options as **create_dir.sh -c -e 12**).

```
---
orchestrate:
  - name: orc_script
    description: creates a local dir
    ansible_options:
      extra_args: -c -e 12
    ansible_script:
      name: scripts/create_dir.sh
    hosts: localhost
    orchestrator: ansible
```

Example 10

Again building on the previous example, you can set more options to the script execution. The script is executed using the script ansible module, so you can set options the script module has. The example below uses the **chdir** option. You can also set other ansible options, and the example below sets the **remote_user** option.

To see all script options see ansible's documentation here.

```
---
orchestrate:
  - name: orc_script
    description: creates a remote dir
    ansible_options:
      remote_user: cloud-user
```

(continues on next page)

(continued from previous page)

```
      extra_args: -c -e 12 chdir=/home
    ansible_script:
      name: scripts/create_dir.sh
    hosts: host01
    orchestrator: ansible
```

Example 11

You have a playbook which needs to run against x number of hosts and requires skipping tasks with a tag set to ssh_auth and requires extra variables.

```
---
orchestrate:
- name: orc_task_auth
  description: "setup key authentication between driver and clients"
  orchestrator: ansible
  hosts: driver
  ansible_playbook:
    name: ansible/ssh_connect.yml
  ansible_options:
    skip_tags:
      - ssh_auth
    extra_vars:
      username: root
      password: redhat
  ansible_galaxy_options:
    role_file: roles.yml
```

Example 12

Example to run playbooks, scripts and shell command as a part of orchestrate task

```
---
- name: orchestrate_1
  description: "orchestrate1"
  orchestrator: ansible
  hosts: localhost
  ansible_playbook:
    name: ansible/list_block_devices.yml

- name: orchestrate_2
  description: "orchestrate2"
  orchestrator: ansible
  hosts: localhost
  ansible_shell:
    - chdir: ./test_sample_artifacts
      command: ls
    - chdir: ./test_sample_artifacts
      command: cp a.txt b.txt
```

(continues on next page)

(continued from previous page)

```
- name: orchestrate_3
  description: "orchestrate3"
  orchestrator: ansible
  hosts: localhost
  ansible_script:
    name: ./scripts/helloworld.py Teflo_user
    executable: python
```

Example 13

Example to use ansible_script with extra arags with in the ansible_script list of dictionary and its paramter in the name field

```
---
- name: orchestrate_1
  description: "orchestrate1"
  orchestrator: ansible
  hosts: localhost
  ansible_script:
    name: ./scripts/helloworld.py Teflo_user
    executable: python
```

Example 14

Example to use ansible_script with extra arags as a part of ansible_options

```
---
- name: orchestrate_1
  description: "orchestrate1"
  orchestrator: ansible
  hosts: localhost
  ansible_script:
    name: ./scripts/helloworld.py Teflo_user
  ansible_options:
    extra_args: executable=python
```

Example 15

Example to use ansible_script and using ansible_options: extra_args to provide the script parameters

```
---
- name: scripta_task
  description: ""
  orchestrator: ansible
  hosts: controller
  ansible_script:
    name: scripts/add_two_numbers.sh
```

(continues on next page)

(continued from previous page)

```
  ansible_options:
    extra_args: X=12 Y=18
```

Example 16

Example to use environment_vars to be passed to the ansible playbook/script/command Variables X and Y are available during the script execution and can be retrieved for additional logic within the script

```
---
- name: scripta_task
  description: ""
  orchestrator: ansible
  hosts: controller
  ansible_script:
    name: scripts/add_two_numbers.sh
  environment_vars:
    X: 12
    Y: 18
```

Resources

For system configuration & product installs use roles from: Oasis Roles

Execute

Overview

Teflo's execute section declares the test execution of the scenario. In most cases there would be three major steps performed during execution:

- cloning the tests
- executing the tests
- gathering the test results, logs, and other important information for the test execution.

The execution is further broken down into 3 different types:

- execution using a command
- execution using a user defined script
- execution using a user defined playbook

The following is the basic structure that defines an execution task, using a command for execution:

```
---
execute:
  - name:
    description:
    executor:
    hosts:
```

(continues on next page)

(continued from previous page)

```
    ignore_rc: False
    git:
      - repo:
        version:
        dest:
    shell:
      - command: cmd_to_execute_the_tests
        chdir:
    artifacts:
      - ~/restraint-example/tests
      - ~/another_artificate_dir
    ansible_options:
```

The following is the basic structure that defines an execution task, using a user defined script for execution:

```
---
execute:
  - name:
    description:
    executor:
    hosts:
    ignore_rc: False
    git:
      - repo:
        version:
        dest:
    script:
      - chdir: /tmp
        name: tests.sh arg1 arg2
    artifacts:
      - ~/restraint-example/tests
      - ~/another_artificate_dir
    ansible_options:
```

The following is the basic structure that defines an exectuion task, using a user defined playbook for execution:

```
---
execute:
  - name:
    description:
    executor:
    hosts:
    ignore_rc: False
    git:
      - repo:
        version:
        dest:
    playbook:
      - name: tests/test.yml
    artifacts:
      - ~/restraint-example/tests
      - ~/another_artificate_dir
```

(continues on next page)

(continued from previous page)

```
    ansible_options:
```

The above code snippet is the minimal structure that is required to create a execute task within teflo. This task is translated into a teflo execute object which is part of the teflo compound. You can learn more about this at the architecture page.

Please see the table below to understand the key/values defined.

Key	Description	Type	Required	Default
name	Name assigned to the execution task	String	Yes	n/a
description	Description of the execution task	String	No	n/a
executor	Name of the executor to be used	String	No	runner
hosts	the machine(s) that execute will run on	String	Yes	n/a
ignore_	ignore the return code of the execution	Boole	No	False
valid_	valid return codes of the execution (success)	list of integers	No	n/a
git	git information for the tests in execution	list of dictionaries	No	n/a
shell	list of shell commands to execute the tests.	list of dictionaries	(Not required; however, one of the following must be defined: shell, script or playbook)	False
script	list of scripts to execute to run the tests.	list of dictionaries	(Not required; however, one of the following must be defined: shell, script or playbook)	False
playbook	list of playbooks that execute the tests.	list of dictionaries	(Not required; however, one of the following must be defined: shell, script or playbook)	False
artifacts	list of all the data to collect after execution. If a directory is listed, all data in that folder will be gathered. A single file can also be listed.	list	No	n/a
artifact_l	A list of data collected during artifacts or a list of additional log files to be considered by Teflo after execution. It is a list of relative path for the directories or files to be considered under the teflo's **.results** folder.	dict	No	n/a
ansible_o	get ansible options for the tests in execution	dictionary	No	n/a
environment_	Additional environment variables to be passed during the test execution	dict	No	environment variables set prior to starting the teflo run are available

Hosts

Teflo provides many ways to define your host. This has already been described in the orchestration section, please view information about defining the hosts here. For more localhost usage refer to the localhost page.

Ansible

The default executor '*runner*' uses ansible to perform the requested actions. This means users can set ansible options to be used by the runner executor. In this case, the options should mostly be used for defining the user that is performing the execution. Please see the following example for more details:

```
ansible_options:
  become: True
  become_method: sudo
  become_user: <become_user>
  remote_user: <remote_user>
```

Note: Teflo uses the ansible copy module to process the results of the requested action. The copy module requires selinux access. Refer to the install guide.

Ansible Logs

To get ansible logs, you must set the **log_path** in the ansible.cfg, and it is recommended to set the **log_filter** in the ansible.cfg as described to filter out non ansible logs. If you do not set the log path or don't provide an ansible.cfg, you will not get any ansible logs. The ansible log will be added to the **ansible_executor** folder under the logs folder of teflo's output, please see Teflo Output for more details.

Return Code for Test Execution

Teflo will fail out if there is a non-zero return code. However, for many unit testing frameworks there is a non-zero return code if there are test failures. For this case, teflo has two options to handle these situations:

1. ignore the return code for the test execution
2. give list of valid return codes that will not flag failure

Option 1 to handle non-zero return codes is called **ignore_rc**, this option can be used at the top level key of execute or can also be used for each specific call. The following shows an example, where it is defined in both areas. The top level is set to False, which is the default, then it is used only for the 2nd pytest execution call, where there are failures:

```
---
execute:
  - name: pytest execution
    description: "execute pytests on the clients"
    hosts: clients
    executor: runner
    ignore_rc: False
    git:
      - repo: https://gitlab.cee.redhat.com/PIT/teflo/pytest-example.git
```

(continues on next page)

(continued from previous page)

```
        version: master
        dest: /home/cloud-user/pytest
    shell:
      - chdir: /home/cloud-user/pytest/tests
        command: python -m pytest test_sample.py --junit-xml test-report/suite1_results.
↪xml
      - chdir: /home/cloud-user/pytest/tests
        command: python -m pytest sample_unittest.py --junit-xml test-report/suite2_
↪results.xml
        ignore_rc: True
    artifacts:
      - /home/cloud-user/pytest/tests/test-report/suite1_results.xml
      - /home/cloud-user/pytest/tests/test-report/suite2_results.xml
```

Options 2 to handle non-zero return codes is called **valid_rc**, this option can also be used at the top level key of execute or can be used for each specific call. If **ignore_rc** is set it takes precedence. The following shows an example, where it is defined in both areas. The top level is set to one value and the call overides it:

```
---
execute:
  - name: pytest execution
    description: "execute pytests on the clients"
    hosts: clients
    executor: runner
    valid_rc: [3, 5, 9]
    git:
      - repo: https://gitlab.cee.redhat.com/PIT/teflo/pytest-example.git
        version: master
        dest: /home/cloud-user/pytest
    shell:
      - chdir: /home/cloud-user/pytest/tests
        command: python -m pytest test_sample.py --junit-xml test-report/suite1_results.
↪xml
      - chdir: /home/cloud-user/pytest/tests
        command: python -m pytest sample_unittest.py --junit-xml test-report/suite2_
↪results.xml
        valid_rc: [2, 7]
    artifacts:
      - /home/cloud-user/pytest/tests/test-report/suite1_results.xml
      - /home/cloud-user/pytest/tests/test-report/suite2_results.xml
...
```

Using Shell Parameter for Test Execution

When building your shell commands it is important to take into consideration that there are multiple layers the command is being passed through before being executed. The two main things to pay attention to are YAML syntax/escaping and Shell escaping.

When writing the command in the scenario descriptor file it needs to be written in a way that both Teflo and Ansible can parse the YAML properly. From a Teflo perspective it is when the the scenario descriptor is first loaded. From an Ansible perspective its when we pass it the playbook we create, cbn_execute_shell.yml, through to the ansible-playbook CLI.

Then there could be further escapes required to preserve the test command so it can be interpreted by the shell properly. From a Teflo perspective that is when we pass the test command to the ansible-playbook CLI on the local shell using the -e "xcmd='<test_command>'" parameter. From the Ansible perspective its when the shell module executes the actual test command using the shell on the designated system.

Let's go into a couple examples

```yaml
shell:
  - command: glusto --pytest='-v tests/test_sample.py --junitxml=/tmp/SampleTest.xml'
             --log /tmp/glusto_sample.log
```

On the surface the above command will pass YAML syntax parsing but will fail when actually executing the command on the shell. That is because the command is not preserved properly on the shell when it comes to the *–pytest* optioned being passed in. In order to get this to work you could escape this in one of two ways so that the *–pytest* optioned is preserved.

```yaml
shell:
  - command: glusto --pytest=\\\"-v tests/test_sample.py --junitxml=/tmp/SampleTest.xml\\
↪\"
             --log /tmp/glusto_sample.log


shell:
  - command: glusto \\\"--pytest=-v tests/test_sample.py --junitxml=/tmp/SampleTest.xml\\
↪\"
             --log /tmp/glusto_sample.log
```

Here is a more complex example

```yaml
shell:
    - command: if [ `echo \$PRE_GA | tr [:upper:] [:lower:]` == 'true' ];
               then sed -i 's/pre_ga:.*/pre_ga: true/' ansible/test_playbook.yml; fi
```

By default this will fail to be parsed by YAML as improper syntax. The rule of thumb is if your unquoted YAML string has any of the following special characters :-{}[]!#|>&%@ the best practice is to quote the string. You have the option to either use single quote or double quotes. There are pros and cons to which quoting method to use. There are online resources that go further into this topic.

Once the string is quoted, you now need to make sure the command is preserved properly on the shell. Below are a couple of examples of how you could achieve this using either a single quoted or double quoted YAML string

```yaml
shell:
    - command: 'if [ \`echo \$PRE_GA | tr [:upper:] [:lower:]\` == ''true'' ];
               then sed -i \"s/pre_ga:.*/pre_ga: true/\" ansible/test_playbook.yml; fi'
```

(continues on next page)

(continued from previous page)

```
shell:
    - command: "if [ \\`echo \\$PRE_GA | tr [:upper:] [:lower:]\\` == \\'true\\' ];
               then sed \\'s/pre_ga:.*/pre_ga: true/\\' ansible/test_playbook.yml; fi"
```

Note: It is NOT recommended to output verbose logging to standard output for long running tests as there could be issues with teflo parsing the output

Extra_args for script and shell

Teflo supports the following parameters used by ansible script and shell modules

Parameters
chdir
creates
decrypt
executable
removes
warn
stdin
stdin_add_newline

Please look here for more info

Ansible Script Module Ansible Shell Module

Using Playbook Parameter for Test Execution

Using the playbook parameter to execute tests works like how playbooks are executed in the Orchestration phase. The only thing not supported is the ability to download roles using the *ansible_galaxy_option*. The following is an example of how run test playbooks.

```
---
execute:
  - name: playbook execution
    description: "execute playbook tests against the clients"
    hosts: clients
    executor: runner
    playbook:
    - name: ansible/test_gather_machine_facts.yml
    ansible_options:
      extra_vars:
        workspace: .
      skip_tags:
       - cleanup
    artifacts:
    - ~/cloud-user/client.facts
```

Note: Unlike the shell or script parameter the test playbook executes locally from where teflo is running. Which means the test playbook must be in the workspace.

Note: extra_vars are set same as the orchestrate stage. Please refer *Extra Vars*

Data Substitution Required for Test Execution

In some cases, you may need to substitute data for the execution. Teflo allows you to substitute the information from the dynamically created hosts.

Let's first take a look at some example data of key/values a user may use for provisioning a host:

```
---
provision:
  - name: test_client_a
    provisioner: openstack-libcloud
    credential: openstack
    image: rhel-7.6-server-x86_64-released
    flavor: m1.small
    networks: [<defined_os_network>]
    floating_ip_pool: "<defined_fip>"
    keypair: pit-jenkins
    groups: clients
    ansible_params:
      ansible_user: cloud_user
      ansible_ssh_private_key_file: <defined_key_file>
```

After the machines are provsioned, we have more information in the host object, and this can be seen by the results.yml file after a provision is successful. Some basic information that is added is the machine's actual name and ip address. The following is what the data looks like after provisioning:

```
---
provision:
  - name: test_client_a
    provisioner: openstack-libcloud
    credential: openstack
    image: rhel-7.6-server-x86_64-released
    flavor: m1.small
    networks: [<defined_os_network>]
    floating_ip_pool: "<defined_fip>"
    keypair: pit-jenkins
    admin_pass: null
    description: null
    files: null
    node_id: 82340e64-c7b7-4a20-a9e3-6511dbc79ded
    security_groups: null
    ip_address: 10.8.250.239
    groups: clients
    ansible_params:
```

(continues on next page)

(continued from previous page)

```
      ansible_user: cloud_user
      ansible_ssh_private_key_file: <defined_key_file>
    data_folder: /var/local/teflo/ljcgm7yl5d
    metadata: {}
...
```

Looking at the data presented above, there is a lot of information about the host, that may be useful for test execution. You can also see the key **metadata**, this key can be used to set any data the user wishes to when running teflo.

The following is an example, where the user plans to use the ip address in an execution command. From the data above, you can see the user is accessing the data from **test_client_a -> ip_address**.

```
---
execute:
  - name: restraint test
    description: "execute tests by restraint framework"
    executor: runner
    hosts: driver
    git:
      - repo: https://gitlab.cee.redhat.com/PIT/teflo/restraint-example.git
        version: master
        dest: ~/restraint-example
    shell:
      - command: /usr/bin/restraint -vvv --host 1=root@{ test_client_a.ip_address }:8081␣
↪--job ./test_sample.xml
        chdir: ~/restraint-example/tests
      - command: /usr/bin/restraint -vvv --host 1=root@{ test_client_b.ip_address }:8081␣
↪--job ./test_sample.xml
        chdir: ~/restraint-example/tests
    artifacts:
      - ~/restraint-example/tests/test_*
```

Artifacts of the Test Execution

After an execution is complete, it is common to get results of the test execution, logs related to the execution, and other logs or files generated by a specific product during the execution. These will all be gathered by teflo and placed in an artifacts directory of your data directory.

For the data gathering, if you specify a folder, teflo will gather all the data under that folder, if you specify a file, it will gather that single file.

The following is a simple example of the data gathering (defining artifacts):

```
---
execute:
...
    artifacts:
      - /home/cloud-user/pytest/tests/test-report/suite1_results.xml
      - /home/cloud-user/pytest/tests/test-report/suite2_results.xml
      - ~/restraint-example/tests/
...
```

Going through the basics of artifacts, the user can archive individual files, as shown by the following example:

```
...
    artifacts:
      - /home/cloud-user/pytest/tests/test-report/suite1_results.xml
      - /home/cloud-user/pytest/tests/test-report/suite2_results.xml
...
```

The user can also collect artifact files using wildcards as shown in the following example:

```
...
    artifacts:
      - /home/cloud-user/pytest/tests/test-report/suite*.xml
...
```

The user can also archive a directory using either of the following two examples:

```
...
    artifacts:
      - ~/restraint-example/tests
...
```

```
...
    artifacts:
      - ~/restraint-example/tests/
...
```

Finally, the user can archive a directory using a wildcard using either of the following two examples:

```
    artifacts:
      - ~/restraint-example/test*
      - ~/restraint-example1/test**
...
```

```
...
    artifacts:
      - ~/restraint-example/test*/
...
```

Teflo by default will **NOT** exit if the collection of artifact task fails. In order to exit the run on an error during collection of artifacts user can set the **exit_on_error** field for executor in the teflo.cfg as below:

```
[executor:runner]
exit_on_error=True
```

Artifact Locations

The **artifact_locations** key is used to keep track of the artifacts that were collected using artifacts key during execute stage. It's a list which consists of the relative path of the artifacts to be considered which are placed under the teflo's **.results** folder. The artifact_locations key is available to users to define locations for artifacts that may not have been collected as part of artifacts but they want to be tracked for later use in Report. The only caveat is the artifacts defined under artifact_locations must be placed in the teflo_data_folder/.results directory. Refer to the *Finding the right artifacts*

Teflo also auto creates the artifacts folder under the .results folder. Users can place their artifacts in this folder as well

In the below example, the payload_dir is the name of the directory which is present under the .results folder

```
---
execute:
  - name:
    description:
    executor:
    hosts:
    ignore_rc: False
    git:
      - repo:
        version:
        dest:
    shell:
      - command: cmd_to_execute_the_tests
        chdir:
    artifacts:
      - ~/restraint-example/tests
      - ~/another_artificate_dir
    artifact_locations:
      - payload_dir/results/
      - payload_dir/results/artifacts/test1.log
```

In the below example, the payload_dir and dir1 are placed in the artifacts folder created by teflo.

```
---
execute:
  - name:
    description:
    executor:
    hosts:
    ignore_rc: False
    git:
      - repo:
        version:
        dest:
    shell:
      - command: cmd_to_execute_the_tests
        chdir:
    artifacts:
      - ~/restraint-example/tests
      - ~/another_artificate_dir
    artifact_locations:
      - artifacts/payload_dir/
```

(continues on next page)

(continued from previous page)

```
      - artifacts/dir1/abc.log
```

Testrun Results for Artifacts collected during the Execute block:

Teflo generates a testrun results summary for all the **xml files** it collects as a part of artifacts OR artifact_locations in an execute block. This summary can be seen in the results.xml as weel as is printed out on the console. The summary shows aggregate summary of all xmls collected and individual summary of each xml file. The summary contains **number of tests passed, failed, errored, skipped and the total tests.**

```
- name: junit
  description: execute junit test on client
  executor: runner
  hosts:
  - laptop
  shell:
  - chdir: /home/client1/junit/tests
    command: javac Sample.java; javac UnitTestRunner.java; javac CustomExecutionListener.
↪java; javac SampleTest.java; java UnitTestRunner SampleTest
  git:
  - repo: https://gitlab.cee.redhat.com/ccit/teflo/junit-example.git
    version: master
    dest: /home/client1/junit
  artifacts:
  - /home/client1/junit/tests/*.log
  - /home/client1/junit/tests/*.xml
  artifact_locations:
    - dir1/junit_example.xml
    - artifacts/client1/junit_example.xml
    - artifacts/client1/ocp_edge_deploment_integration_results.xml
    - artifacts/client1/SampleTest.xml
  testrun_results:
    aggregate_testrun_results:
      total_tests: 22
      failed_tests: 9
      error_tests: 0
      skipped_tests: 0
      passed_tests: 13
    individual_results:
    - junit_example.xml:
        total_tests: 6
        failed_tests: 2
        error_tests: 0
        skipped_tests: 0
        passed_tests: 4
    - junit_example.xml:
        total_tests: 6
        failed_tests: 2
        error_tests: 0
        skipped_tests: 0
        passed_tests: 4
```

(continues on next page)

(continued from previous page)

```
    - ocp_edge_deploment_integration_results.xml:
        total_tests: 8
        failed_tests: 5
        error_tests: 0
        skipped_tests: 0
        passed_tests: 3
    - SampleTest.xml:
        total_tests: 2
        failed_tests: 0
        error_tests: 0
        skipped_tests: 0
        passed_tests: 2
```

This is the default behavior of Teflo. If a user does not want this summary generated, user can change the following setting to False in the teflo.cfg

```
[executor:runner]
testrun_results=False
```

Note: Teflo expects the xmls collected to have the **<testsuites>** tag OR **<testsuite>** as its root tag, else it skips those xml files for testrun summary generation

Using environment variables:

In the below example the environment variables data_dir and uname are made available during the playbook execution

```
---
execute:
  - name: playbook execution
    description: "execute playbook tests against the clients"
    hosts: clients
    executor: runner
    playbook:
    - name: ansible/test_gather_machine_facts.yml
    ansible_options:
      extra_vars:
        workspace: .
      skip_tags:
       - cleanup
    artifacts:
    - ~/cloud-user/client.facts
    environment_vars:
      data_dir: /home/data
      uname: teflo_user
```

Common Examples

Please review the following for detailed end to end examples for common execution use cases:

- Pytest Example
- JUnit Example
- Restraint Example

Report

Overview

Teflo's report section declares which test artifacts collected during execution are to be imported into a Report & Analysis system. The input for artifact import will depend on the destination system.

First, let's go over the basic structure that defines a Report resource.

```
---
report:
  - name: <name>
    description: <description>
    executes: <execute>
    importer: <importer>
```

Important: The Reporting systems currently supported are Polarion and Report Portal. These systems can be accessed using teflo's plugins **teflo_polarion_plugin** and **teflo_rppreproc_plugin** These plugins are only available for internal RedHat use at this time. Users can put in tickets here for new plugin development or contribute towards this effort. Please refer Developers Guide on how to contribute towards the plugin.

Key	Description	Type	Required
name	The name of the test artifact to import. This can be a full name of an artifact, a shell pattern matching string, or a string using Teflo's data-passthru mechanism	Strin;	True
description	A description of the artifact being imported	Strin;	False
executes	The name of the execute block that collected the artifact.	List	False
importer	The name of the importer to perform the import process.	Strin;	True

Executes

Defining a Teflo execute resource is optional. Teflo uses the execute resource for two reasons:

- It uses the **artifact_locations** key as a quick way to check if the artifact being requested was collected and where to find it.
- It uses the Asset resources assigned to the Execute to perform the internal templating if a data-passthru string is being used in the name key as search criteria.

Finding the right artifacts

As noted in the table, the driving input will be the name key. The name can be a string defining the exact file/folder name, a shell matching pattern, or a teflo data-passthru pattern. Depending on the pattern used it will narrow or widen the search scope of the search. How teflo performs the search is by the following

- Check if an execute resource was defined with the **execute** and then check **artifact_locations** key is defined for the execute in the execute section.
- If there is an **execute** key and the artifact is listed as an item that was collected in the **artifact_locations** key, teflo will immediately validate the location.
- If no **execute** key is defined, or an execute with no **artifact_location** key is used, or the artifacts is not shown as one of the items contained in the the artifact_location key, or the item location in the artifact_location key is no longer valid, it proceeds to walk the *data_folder/.results* folder.
- If no artifacts are found after walking the *data_folder/.results*, teflo will abort the import process.
- If artifacts are found, the list of artifacts will be processed and imported into the respective reporting system.

More information on artifact_locations key refer *Finding Locations*

Notification

Overview

Teflo's notification section declares what messages are to be sent and to whom when triggered. The current notification mechanism is *email*.

First lets go over the basic structure that defines a notification task.

```
---
notifications:
  - name: test_email
    notifier: email-notifier
    credential: email
    on_success: true
    to:
      - jsmith@redhat.com
    from: qe-rh@redhat.com
```

The above code snippet is the minimal structure that is required to create a notification task within teflo. This task is translated into a teflo notification object which is part of the teflo compound. You can learn more about this at the architecture page. Please see the table below to understand the key/values defined.

Key	Description	Type	Re-quired	Default
name	The name of the notification to define	String	Yes	n/a
de-scrip-tion	A description of what the notification is trying to accomplish	String	No	n/a
noti-fier	The notifier to use to send notifications when triggered above	String	Yes	email-notifier
on_start	trigger to send a notification when a task is going to be executed	Boolea	No	False
on_succe	trigger to send a notification when a task has executed successfully	Boolea	No	True
on_failu	trigger to send a notification when a task has executed unsuccessfully	Boolea	No	True
on_dema	disable automatic trigger of the notification. Must be manually triggered	Boolea	No	False
on_tasks	Filter for which tasks should trigger a notification	List	No	All Tasks (Validate, Provision, Orchestrate, Execute, Report, Cleanup)

Triggers

By default, Teflo implicitly triggers on both **on_success** and **on_failure** for all completed task types. If you would like to set it for either/or, you can explicitly set either parameter to **true**.

If you would like to have teflo trigger notifications before the start of a task rather than after, you can set **on_start** to **true**. The **on_start** option is mutually exclusive to **on_success/on_failure**.

If you would like to have teflo not trigger notifications automatically and you would like to control when to trigger notifications in your workflow, you can set the **on_demand** flag to true.

If you would like to filter so that only certain tasks trigger notifications, you can set **on_tasks** to a list of any combination of supported teflo tasks. This does not apply to **on_demand**.

There are further capabilities to controlling the triggering of any notifications from the command line.

For example, if you have defined different notifications in your scenario with different triggers but are interested in triggering certain ones for a particular run, you can specify which ones to skip using the **–skip-notify** option

```
teflo run -s scenario.yml -w . -t provision --skip-notify notification_a --skip-notify␣
↪notification_b
```

If you would like to temporarily disable triggering notifications for the entire scenario for a particular run without permanently setting them to **on_demand**. You can use the **–no-notify** option

```
teflo run -s scenario.yml -w . -t execute -t report --no-notify
```

Sending Email Notifications

Credentials/Configure

To configure the email notification, you will need to have your SMTP configuration in your teflo.cfg file, see SMTP Configuration for more details.

Email

The following shows all the possible keys for defining an email notification using the **email-notifier** notifier:

```
---
notifications:
  - name: <name>
    notifier: <notifier>
    to: <list_of_values>
    from: <from>
    cc: <list_of_values>
    subject: <subject>
    attachments: <list_of_values>
    message_body: <multiline value>
    message_template: <template_path>
```

Key	Description	Type	Required
to	A list of email addresses that this notification should be sent to.	List	True
from	The email address the notification should be from.	String	True
cc	The list of email addresses that you want to send teflo copies to.	List	False
subject	The subject of the message that should be included.	String	False
attachments	List of attachments to include when the message is sent.	List	False
message_body	The text body of the message to include overriding Teflo's default message template.	String	False
message_templat	A relative path to a text email body template in Teflo's workspace that should be used. It overrides Teflo's default message template.	String	False

Message Content

Teflo has a default messaging template that is sent when no **message_body** or **message_template** parameter is used. Teflo uses some internal data about the tasks performed by the scenario. Below is the list of data being rendered into the message

- overall status of the Scenario execution
- The list of Teflo tasks that passed and/or failed
- The list of artifacts that were collected after test execution if any

- The import result urls of any test artifacts that were imported into a reporting system

Teflo makes its scenario and scenario_graph objects available to user when designing their own messaging template. The key for teflo's scenario object is **scenario** and for scenario_graph is **scenario_graph**

Along with the scenario object, users can get all the variables set during teflo run as well as environment variables as scenario_vars dictionary to be used in the templates. The key for this is **scenario_vars**

Examples

Let's go into some examples of you can define your notification resources

Example 1

You want to trigger a notification on all successful tasks using the default template

```
---
notifications:
  - name: test_email
    notifier: email-notifier
    credential: email
    on_success: true
    to:
      - jsmith@redhat.com
    from: qe-rh@redhat.com
```

Example 2

You want to trigger a notification before the start of all tasks using a messaging template

```
---
notifications:
  - name: msg_template
    notifier: email-notifier
    credential: email
    on_start: true
    to:
      - jsmith@redhat.com
      - fbar@redhat.com
    from: qe-team@redhat.com
    subject: test email notification using default template {{ UUID }}
    message_template: email_templ.txt
```

Teflo's scenario data could be used to format the template email_templ.txt as shown in the examples below:

a.

```
Hello All,

This is a Teflo Notification.
```

(continues on next page)

(continued from previous page)

```
Teflo scenario, {{ scenario.name }}, has provisioned  the asset:

{{ scenario.assets[0].name }}

The data directory is {{ scenario_vars.TEFLO_DATA_FOLDER }}
```

b.

```
Hello All,

This is a Teflo Notification for Execute task.

 {% if scenario.get_executes() %}
     {% for execute in scenario.get_executes() %}
     Execute task name: {{ execute.name }}
             {% if execute.artifact_locations %}
     Collected the following artifacts:
             {% for file in execute.artifact_locations %}
             - {{ file }}
             {% endfor %}
         {% endif %}
         {% if execute.testrun_results %}

     These are the test results of the scenario:

       Total Tests: {{ execute.testrun_results.aggregate_testrun_results.total_tests }}
       Passed Tests: {{ execute.testrun_results.aggregate_testrun_results.passed_tests }}
       Failed Tests: {{ execute.testrun_results.aggregate_testrun_results.failed_tests }}
       Skipped Tests: {{ execute.testrun_results.aggregate_testrun_results.skipped_tests␣
↪}}

         {% endif %}
     {% endfor %}
 {% else %}
  No execute tasks were run
 {% endif %}
```

This is how the email sent using above template will read:

```
Hello All,

 This is a Teflo Notification for Execute task.

      Execute task name: Test running playbook
      Collected the following artifacts:
              - artifacts/localhost/rp_preproc_qmzls.log
              - artifacts/localhost/junit_example_5_orig.xml

      These are the test results of the scenario:

        Total Tests: 6
        Passed Tests: 4
```

(continues on next page)

(continued from previous page)

```
        Failed Tests: 2
        Skipped Tests: 0

      Execute task name: Execute2
      Collected the following artifacts:
              - artifacts/localhost/rp_preproc_qmzls.log
              - artifacts/localhost/junit_example_5_orig.xml

      These are the test results of the scenario:

        Total Tests: 6
        Passed Tests: 4
        Failed Tests: 2
        Skipped Tests: 0
```

Example 3

You want to trigger a notification regardless on failures of the Validate and Provision task but you want to include a multiline string in the descriptor file.

```
---
notifications:
  - name: msg_body_test
    notifier: email-notifier
    credential: email
    on_failure: true
    on_tasks:
      - validate
      - provision
    to: [jsnith@redhat.com, fbar@redhat.com]
    from: qe-team@redhat.com
    subject: test notification using message body.
    message_body: |
      Hello All,

      This is a Teflo Test notification. For Jenkins Job {{ Job }}.

      Thanks,

      Waldo
```

Example 4

You want to trigger a notification regardless only on failures of all tasks using the default template message but you want to include a file as an attachment.

```
---
notifications:
  - name: msg_test
    notifier: email-notifier
    credential: email
    on_failure: true
    to: [jsnith@redhat.com, fbar@redhat.com]
    from: qe-team@redhat.com
    subject: test notification using message attachments.
    attachments:
      - workpsace/folder/file.txt
```

Example 5

You don't want a notification to trigger automatically.

```
---
notifications:
  - name: msg_test
    notifier: email-notifier
    credential: email
    on_demand: true
    to: [jsnith@redhat.com, fbar@redhat.com]
    from: qe-team@redhat.com
    subject: test notification only when manually triggered.
```

Example 6

Using custom template and using teflo's data for formatting

```
---
notifications:
  - name: msg_template
    notifier: email-notifier
    credential: email
    on_start: true
    to:
      - jsmith@redhat.com
      - fbar@redhat.com
    from: qe-team@redhat.com
    subject: test email notification using default template {{ UUID }}
    message_template: email_templ.txt
```

Example 7

Using custom template and using teflo's variables for formatting

Consider teflo_var.yml is the file set as the default variable file in teflo.cfg

```
[defaults]
var_file=./teflo_var.yml
```

The contents of teflo_var.yml:

```
---
username: teflo_user
msg_template: template.jinja
var_a: hello
```

The template template.jinja will look like this

```
{{scenario_vars.var_a}} {{ scenario_vars.username }},

This is a Teflo Notification.

Teflo has completed executing the scenario, {{ scenario.name }}, with overall status:

{% if scenario.overall_status == 0 %}
Passed
{% else %}
Failed
{% endif %}

The data folder is {{ scenario_vars.TEFLO_DATA_FOLDER }}
```

Scenario file notification block

```
---
notifications:
  - name: msg_template
    notifier: email-notifier
    credential: email
    on_tasks: ['provision']
    to:
      - jsmith@redhat.com
      - fbar@redhat.com
    from: qe-team@redhat.com
    subject: test email notification is for user {{ username }}
    message_template: {{ msg_template }}
```

The above example post run will be seen as following in the results.yml file, where the variables from teflo_var.file are used

```
notifications:
  - name: msg_template
    notifier: email-notifier
    credential: email
```

(continues on next page)

(continued from previous page)

```
    on_success: true
    on_failure: true
    on_tasks:
      - provision
    on_start: false
    on_demand: false
    to:
      - jsmith@redhat.com
      - fbar@redhat.com
    from: qe-team@redhat.com
    subject: test email notification is for user teflo_user
    message_template: template.jinja
```

The above example will send email which will look like this:

```
hello teflo_user,

This is a Teflo Notification.

Teflo has completed executing the scenario, test1, with overall status:
Passed

The data folder is /home/workspace/teflo/data_folder/nzohposc6v/
```

Sending Chat Notifications

Teflo_webhooks_notification_plugin allows users to send chat notification during and/or post teflo run. To get more information about this plugin ,on how to install and use it please visit teflo_webhooks_notification_plugin

Timeout settings for Teflo Tasks

This feature will allow users to set a time limit for all the teflo tasks. This can be done in either of the following two ways

1. defining the **timeout** fields in teflo.cfg. These values will be applied throughout the scenario descriptor file:

   ```
   [timeout]
   provision=500
   cleanup=1000
   orchestrate=300
   execute=200
   report=100
   validate=10
   ```

2. defining the **timeout** fields in SDF. Here you can define below timeouts for individual task blocks within the SDF:

   validate_timeout, provision_timeout, orchestrate_timeout, execute_timeout, report_timeout, cleanup_timeout, notification_timeout

```
---
name: example
description: An example scenario for timeout

provision:
- name: test
    group: client
    provisioner: linchpin-wrapper
    provider:
    name: openstack
    credential: openstack
    image: rhel-7.4-server-x86_64-released
    flavor: m1.small
    keypair: {{ key }}
    networks:
        - provider_net_cci_4
    ansible_params:
    ansible_user: cloud-user
    ansible_ssh_private_key_file: /home/junqizhang/.ssh/OCP
    # you define provision_timeout, orchestrate_timeout, cleanup_timeout,␣
↪report_timeout here from SDF
    provision_timeout: 200

report:
- name: SampleTest.xml
    description: import results to polarion
    executes: junitTestRun
    provider:
    credential: polarion-creds
    name: polarion
    project_id: Teflo1
    testsuite_properties:
        polarion-lookup-method: name
        polarion-testrun-title: e2e-tests
    report_timeout: 120
```

Note: If the timeout values are defined from SDF, it will overwrite the timeout values defined from teflo.cfg

When we put all of these sections together, we have a complete scenario to provide to teflo. You can see an example of a complete scenario descriptor below:

```
---

# template used to demonstrate the layout of a multi product (interop)
# scenario definition consumable by the interop framework in this case
# teflo.

# generic section

# defines common information about the scenario

```

(continues on next page)

(continued from previous page)

```
name: demo
description: >
    Scenario to demonstration the teflo framework.

# resource checking section

# As part of the validation that teflo performs, it can also
# check the status of resources that an end to end scenario
# relies on.  The user can set a list of services
# that need to be checked for status
# prior to the start of the teflo workflow under monitored_services.
#
# Note: these services will only be validated if you
# set the resource_check_endpoint key in your teflo.cfg file e.g.
#
# in teflo.cfg add:
# [defaults]
# resource_check_endpoint=<endpoint url>
# Currently only semaphore and statuspage.io is supported
#
# Along with services, users can run their own validation playbooks or scripts before
# starting the teflo workflow. If the validation here fails the workflow doe snot move␣
↪ahead
# Playbooks and scripts use Teflo's ansible executor/runner

resource_check:
  monitored_services:
    - ci-rhos
    - brew
    - polarion
    - umb
    - errata
    - rdo-cloud
    - covscan
    - rpmdiff
    - gerrit.host.prod.eng.bos.redhat.com
    - code.engineering.redhat.com
  playbook:
    - name: ansible/list_block_devices.yml
      ansible_options:
        extra_vars:
          X: 18
          Y: 12
          ch_dir: ./scripts/
    - name: ansible/tests/test_execute_playbook.yml
      ansible_options:
        extra_vars:
          X: 12
          Y: 12
          ch_dir: ../../scripts/
  script:
      - name: ./scripts/hello_world1.py Teflo_user
```

(continues on next page)

(continued from previous page)

```
        executable: python
      - name: ./scripts/add_two_numbers.sh X=15 Y=15

# include section

# Defines any other scenario files that need to be executed
# These scenario files have to be in the same workspace as the master/original scenario
# During running of teflo workflow tasks from the included scenarios will be run
# e.g if task selected is provision, the resources of master scenario will be provisioned
# followed by provisioning of resources defined in the included scenarios

include:
  - py3_incl_prov.yml
  - py3_incl_orch.yml

# provision section

# defines all systems required for the scenario

provision:
  # test driver
  - name: testdriver                              # machine name used for creation
    description: "test driver"                    # describes the purpose of the host
    provisioner: openstack-libcloud               # provisioner being used to provision␣
↪the asset
    credential: openstack                         # credentials to authenticate the␣
↪openstack instance
    image: rhel-7.5-server-x86_64-released        # image to boot instance based on
    flavor: m1.small                              # instance size
    networks:                                     # instance internal network
      - <internal-network>
    floating_ip_pool: "10.8.240.0"                # instance external network
    keypair: <keypair>                            # instance ssh key pair
    groups: testdriver                            # host group
    ansible_params:                               # defines ansible parameters for␣
↪connection
      ansible_user: root
      ansible_ssh_private_key_file: <private-key-filename>

  # test client 1
  - name: testclient01                            # machine name used for creation
    description: "test client 01"                 # describes the purpose of the host
    provisioner: openstack                        # provisioner to create host using␣
↪openstack
    credential: openstack                         # credentials to authenticate the␣
↪openstack instance
    image: rhel-7.5-server-x86_64-released        # image to boot instance based on
    flavor: m1.small                              # instance size
    networks:                                     # instance internal network
      - <internal-network>
    floating_ip_pool: "10.8.240.0"                # instance external network
      keypair: <keypair>                          # instance ssh key pair
```

(continues on next page)

(continued from previous page)

```
    groups: testclient                            # host group
    ansible_params:                               # defines ansible parameters for␣
↪connection
      ansible_user: root
      ansible_ssh_private_key_file: <private-key-filename>

  # test client 2, defining a static machine
  # this is useful if you wish to skip teflo's provisioning
  # this machine can be referenced in orchestrate and execute
  - name: testclient02                            # machine name used for creation
    description: "test client 02"                 # describes the purpose of the host
    ip_address: <machine_ip_address>
    groups: testclient                            # host group
    ansible_params:                               # defines ansible parameters for␣
↪connection
      ansible_user: root
      ansible_ssh_private_key_file: <private-key-filename>


# orchestrate section

# defines all actions to be performed for the scenario. these actions will be
# executed against the systems defined in the provision section. Each action
# will define which system to run against.
# Then, three types of orchestrate actions are supported by the default orchestrator␣
↪(ansible):
# 1. ansible_shell
# 2. ansible_script
# 3. ansible_playbook
# Only one type of orchestrate action can be run per action.

orchestrate:
  # user defined playbook to execute
  - name: task_1                                  # action name
    description: "performs custom config"         # describes what is being performed on␣
↪the hosts
    orchestrator: ansible                         # orchestrator module to use in this␣
↪case ansible
    hosts:                                        # hosts which the action is executed on
      - testclient01                              # ref above ^^: provision.testclient01
      - testclient02                              # ref above ^^: provision.testclient02
    ansible_playbook:                             # using ansible playbook
      name: custom.yml                            # name (playbook name) (full filename␣
↪and path relative to the workspace)
    ansible_options:                              # options used by ansible orchestrator
      extra_vars:
        var01: value01
    ansible_galaxy_options:                       # options used by ansible galaxy
      role_file: role.yml

  # create ssh key pair for ssh connection between driver/client(s)
  - name: create_ssh_keypair                       # action name
```

(continues on next page)

(continued from previous page)

```
    description: "creates ssh key pair for auth" # describes what is being performed on␣
↪the hosts
    orchestrator: ansible                        # orchestrator module to use in this␣
↪case ansible
    hosts:                                       # hosts which the action is executed on
      - testdriver                               # ref above ^^: provision.testdriver
    ansible_playbook:                            # playbook name(full filename and path␣
↪relative to the workspace)
      name: create_ssh_keypair.yml
    ansible_options:                             # options used by ansible orchestrator
      extra_vars:
        user: root

  # inject driver ssh public key pair to client(s)
  - name: inject_pub_key                        # action name
    description: "injects ssh keys into sut"    # describes what is being performed on␣
↪the hosts
    orchestrator: ansible                       # orchestrator module to use in this␣
↪case ansible
    hosts:                                      # hosts which the action is executed on
      - testdriver                              # ref above ^^: provision.testdrive
    ansible_playbook:
      name: inject_pub_key.yml                  # playbook name(full filename and path␣
↪relative to the workspace)
    ansible_options:                            # options used by ansible orchestrator
      extra_vars:
        user: root
        machine:
          - testclient01
          - testclient02

  - name: rhn_subscribe                         # action name
    description: "subscribe to rhsm"            # describes what is being performed on␣
↪the hosts
    orchestrator: ansible                       # orchestrator module to use in this␣
↪case ansible
    hosts:                                      # hosts which the action is executed on
      - all                                     # ref above ^^ to all hosts : provision.*
    ansible_playbook:
      name: rhn_subscribe.yml                   # playbook name(full filename and path␣
↪relative to the workspace)
    ansible_options:                            # options used by ansible orchestrator
      extra_vars:
        rhn_hostname: subscription.rhsm.stage.redhat.com
        rhn_user: rhel_server_01
        rhn_password: password
        auto: True

   # use FQCN and collection install
  - name: Example 1                             # action name
    description: "use fqcn"                     # describes what is being performed on␣
↪the hosts
```

(continues on next page)

(continued from previous page)

```
    orchestrator: ansible                         # orchestrator module to use in this␣
↪case ansible
    hosts:                                        # hosts which the action is executed on
      - all                                       # ref above ^^ to all hosts : provision.*
    ansible_playbook:
      name: namespace.collection1.playbook1         # playbook name(Using FQCN)
    ansible_galaxy_options:
      role_file: requirements.yml                 # A .yml file to describe␣
↪collection(name,type,version)

# execute section

# defines all the tests to be executed for the scenario
# Each execute task has an option to clone a git,
# where the tests resides if not done in orchestrate
# Then, three types of execution supported by the default executor (runner):
# 1. shell
# 2. script
# 3. playbook
# One must be selected
# Finally, each task has an optional artifacts key used for
# data gathering after the test execution.

execute:
  - name: test_suite_01
    description: "execute tests against test clients"
    executor: runner
    hosts: driver
    git:
      - repo: https://server.com/myproject.git
        version: test-ver-0.1
        dest: /tmp
    shell:
      - chdir: /tmp
        command: /usr/bin/restraint --host 1={{testclient01}}:8081 --job foo.xml
    artifacts: retraint-*, test.log

  - name: test_suite_02
    description: "execute tests against test clients"
    executor: runner
    hosts: driver
    git:
      - repo: https://server.com/myproject.git
        version: test-ver-0.1
        dest: /tmp
    script:
      - chdir: /tmp
        name: tests.sh arg1 arg2
    artifacts: retraint-*, test.log

  - name: test_suite_03
    description: "execute tests against test clients"
```

(continues on next page)

(continued from previous page)

```
    executor: runner
    hosts: driver
    git:
      - repo: https://server.com/myproject.git
        version: test-ver-0.1
        dest: /tmp
    playbook:
      - chdir: /tmp
        name: test.yml
    artifacts: retraint-*, test.log

# report section

# Teflo supports importing to external tools using teflo plugins. Please refer the␣
↪report section
# under Scenario Descriptor for more information on plugins
# Below example is for importing test runs xmls to Polarion

report:
    - name: suite1_results.xml                           # pattern to match the xml file␣
↪to be imported
      description: import suite1 results to polarion     # description of the reporting␣
↪task
      executes: test_suite_01                            # execute task to look for the␣
↪artifacts/xml mentioned under name
      importer: polarion                                 # importer to be used
      credential: polarion-creds                         # credentials to connect to the␣
↪external tool(provided under teflo.cfg)


# notification tasks

# Teflo supports notification using email as default. Please refer notification section␣
↪under
# scenario descriptor to know more about notification using webhook plugins and␣
↪different triggers
# that can be set for notification
# Below example is for notification using email on completion of provision task

notifications:
  - name: notify1                                         # task name
    notifier: email-notifier                              # notifier to be used
    credential: email                                     # credentials needed for␣
↪notifier to be set under teflo.cfg
    on_tasks:                                             # trigger for notification to␣
↪be sent
      - provision
    to:                                                   # list of email addresses to␣
↪send the notification to
      - abc@redhat.com
      - pqr@redhat.com
      - xyz@redhat.com
```

(continues on next page)

(continued from previous page)

```
    from: team@redhat.com                                 # email the notification is␣
↪from
    subject: 'Provision task completed'                   # subject of the email
```

2.4.2 Data Pass-through

This topic focuses on how you can pass data between different tasks of teflo.

Orchestrate

Teflo's orchestrate task is focused with one goal, to configure your test environment. This configuration consists of install/configure products, test frameworks, etc. This is all defined by the user. When you setup your scenario descriptor file (SDF) orchestrate section, you define the actions you wish to run. Lets look at an example below:

```
---

name: orchestrate example
description: orchestrate example showing data pass through

provision:
  - name: localhost
    groups: local
    ip_address: 127.0.0.1
    ansible_params:
      ansible_connection: local

orchestrate:
  - name: orc_task1
    description: "Install product a."
    orchestrator: ansible
    hosts: localhost
    ansible_playbook:
      name: ansible/install_product_a.yml
    cleanup:
      name: ansible/post_product_a_install.yml
      description: "Perform post product a tasks prior to deleting host."
      orchestrator: ansible
      hosts: localhost

  - name: orc_task2
    description: "Install product b using data from product a."
    orchestrator: ansible
    hosts: localhost
     ansible_playbook:
       name: ansible/install_product_b.yml
```

The orchestrate section above has two actions to be run. Both actions are ansible playbooks. This example shows installing product a then product b. Where product b requires return data from the installation of product a. How can the second playbook installing product b get the return data from product a playbook? The recommended way for this is to write return data from product a to disk. This would make the data persistent since when a playbook exits, anything wrote to memory goes out of scope. Using ansible custom facts to write to disk allows the second playbook to have access to the return data from product a install.

This example can be found at the following page for you to run.

Note: Please note the example uses localhost so everything is wrote to the same machine. In a real use case where you want to access the data from a secondary machine where product b is installed. Inside your playbook to install product b, you would want to have a task to delegate to the product a machine to fetch the return data needed.

An optional way to pass data through from playbook to playbook is to have one master playbook. Inside the master playbook you could have multiple plays that can access the return data from various roles and tasks. Please note that this way is not recommended by teflo. Teflos scenario descriptor file allows users with an easy way to see all the configuration that is performed to setup the environment. With having multiple playbooks defined under the orchestrate section. It makes it easier to understand what the scenario is configuring. When having just one action defined calling a master playbook. It then requires someone to go into the master playbook to understand what actions are actually taking place.

There are cases where you want to pass some data about the test machines as a means of starting the configuration process rather during the configuration process. For example, say you've tagged the test machine with metadata that would be useful to be used in the configuration as extra variables or extra arguments. Teflo has the ability to template this data as parameters. Let's take a look at a couple examples:

```yaml
---
provision:
  - name: host01
    groups: node
    provider:
      name: openstack
      ...
    ip_address:
      public: 1.1.1.1
      private: 192.168.10.10
    metadata:
      key1: 'value1'
      key2: 'value2'
      ...
    ansible_params:
      ansible_user: cloud-user
      ...

orchestrate:
  - name: orc_playbook
    description: run configure playbook and do something with ip
    orchestrator: ansible
    hosts: host01
    ansible_playbook:
      name: ansible/configure_task_01.yml
    ansible_options:
      extra_vars:
        priv_ip: <NEED PRIVATE IP OF HOST>

  - name: orc_script
    description: run configure bash script and do something with metadata
```

(continues on next page)

(continued from previous page)

```
    orchestrator: ansible
    hosts: host01
    ansible_script:
      name: scripts/configure_task_02.sh
    ansible_options:
      extra_args: X=<NEED METADATA> Y=<NEED METADATA>
```

We have two orchestrate tasks, one wants to use the private ip address of the machine to configure something on the host. The other wants to use some metadata that was tagged in the test resource. Here is how you could do that

```
---
provision:
  <truncated>

orchestrate:
  - name: orc_task1
    description: run configure playbook and do something with ip
    orchestrator: ansible
    hosts: host01
    ansible_playbook:
      name: ansible/configure_task_01.yml
    ansible_options:
      extra_vars:
        priv_ip: '{ host01.ip_address.private }'

  - name: orc_task2
    description: run configure bash script and do something with metadata
    orchestrator: ansible
    hosts: host01
    ansible_script:
      name: scripts/configure_task_02.sh
    ansible_options:
      extra_args: X={ host01.metadata.key1 } Y={ host01.metadata.key2 }
```

Teflo will evaluate these parameters and inject the correct data before passing these on as parameters for Ansible to use.

Note: extra_vars used under ansible_options is a dictionary , hence the value being injected needs to be in single or double quotes else data injection will not take place e.g. '{ host01.ip_address.private }' or "{ host01.ip_address.private }"

Execute

Teflo's execute task is focused with one goal, to execute the tests defined against your configured environment. Some tests may require data about the test machines. The question is how can you get information such as the IP address as a parameter option for a test command? Teflo has the ability to template your test command prior to executing it. This means it can update any fields you set that require additional data. Lets look at an example below:

```
---
provision:
  - name: driver01
    groups: driver
    provider:
      name: openstack
      ...
    ip_address: 0.0.0.0
    metadata:
      key1: value1
      ...
    ansible_params:
      ansible_user: cloud-user
      ...

  - name: host01
    groups: node
    provider:
      name: openstack
      ...
    ip_address: 1.1.1.1
    metadata:
      key1: value1
      ...
    ansible_params:
      ansible_user: cloud-user
      ...

execute:
  - name: test
    executor: runner
    hosts: driver
    shell:
      - command: test_command --host <NEEDS_IP_OF_HOST01>
```

The above example has two test machines and the one of the tests requires a parameters of the host01 machine ip address. This can easily be passed to the test command using templating. Lets see how this is done:

```
---
provision:
<truncated>

execute:
  - name: test
    executor: runner
    hosts: driver
```

(continues on next page)

(continued from previous page)

```
    shell:
      - command: test_command --host { host01.ip_address }
```

As you can see above you can reference any data from the host resource defined above. You could also access some of the metadata for the test.

```
---
provision:
<truncated>

execute:
  - name: test
    executor: runner
    hosts: driver
    shell:
      - command: test_command --host { host01.ip_address } \
        --opt1 { host01.metadata.key1 }
```

Teflo will evaluate the test command performing the templating (setting the correct data) and then executes the command. These are just a couple examples on how you can access data from the host resources defined in the provision section. You have full access to all the key:values defined by each host resource.

Note: if the instance has been connected to multiple networks and you are using the linchpin-wrapper provisioner, the ip addresses assigned to the instance from both networks will be collected, and stored as a dictionary. Hence, to use the data-passthrough in this situation you would do something like the following:

{ host01.ip_address.public }

2.4.3 Teflo Output

Data Folder

When you call teflo, all runtime files, logs, artifacts, etc get stored within a data folder. By default teflo sets the base data folder as the */tmp* directory. This can be overridden by either the command line option or within the *teflo.cfg*. Each teflo run creates a unique data folder for that specific run. This unique data folder is stored at the parent data folder mentioned above. With having this unique data folder per teflo run. It allows you to keep those logs for that specific run for historical purposes. These unique data folders are based on a UUID. You must be thinking, how can I reference a data folder to my given run? To help easily find the last execution data folder, teflo provides an additional folder for easily accessing this. At the end of each teflo run, a new directory named *.results* will be created at the data folder you supplied to teflo. This directory is a exact copy of the data folder for the last given teflo run. It allows you to easily access the last runs files.

Results File

Each time you call teflo, you supply it a scenario descriptor file. This file tells teflo exactly what it should do. After teflo finishes running the task it was supplied with it needs to potentially update the scenario descriptor file with additional information from that given run. Instead of modifying the input file, teflo creates a new scenario descriptor file (an exact copy of the input one) just with additional information from the run. For example: the provision task finishes and has additional data about the host machines created. The results file would include this additional data returned back from the provision task. This file is stored in the data folder *.results* directory named *{scenario descriptor file name without file extension}_results.yml*. This allows users to continue executing teflo tasks if run individually. It eliminates the need for restarting the entire scenario from the beginning.

Below is an example emphasizing on the additional data added back to the scenario descriptor file after a successful provision task run:

```
...
name: ffdriver
provider:
  credential: openstack
  flavor: m1.small
  floating_ip_pool: <definied ip pool>
  hostname: ffdriver_l3zqh
  image: rhel-7.5-server-x86_64-released
  keypair: pit-jenkins
  name: openstack
  networks:
  - pit-jenkins
  node_id: 4beb3789-1e61-4f7c-bf9e-722ed480b280
ip_address: 10.8.249.2
...
```

Included Scenario Results File

If *include* section is present in the scenario file and it has a valid scenario descriptor file, then on a teflo run there will be an additional results file for this included scenario with its filename(without file extension) in the prefix. e.g. common_results.yml will be the name of the results file for included scenario with file name common.yml. This allows the users to use this common_results.yml file and include it in other scenarios as needed, reducing the execution time and code duplication. The included scenario results file is also located in the .results folder where results.yml is stored

Results Folder

As mentioned above in the data folder section, at the end of each teflo run a *.results* directory is created with the latest results for a given run. You will find a variety of directories/files here. Lets go over some of the common ones you will see.

Note: Each teflo task can produce different files that will be archived in the results directory. For example: when running the orchestrate task you would see an inventory directory created along with an ansible log. During the execute task you would see a directory for artifacts containing results produced from automated test suites.

```
.results/
├── artifacts
```

(continues on next page)

(continued from previous page)

```
│     ├── client-a
│     │   ├── suite1_results.xml
│     │   └── suite2_results.xml
│     └── client-b
│         ├── suite1_results.xml
│         └── suite2_results.xml
├── inventory
│     └── inventory_uuid
├── logs
│     ├── ansible_executor
│     │   └── ansible.log
│     └── ansible_orchestrator
│         └── ansible.log
├──  <scenaio_filename_without_file_extension>_results.yml
└──  results.yml
```

Name	Description	Type
artifacts	A directory containing all artifacts generated by the given tests stored in sub directories named by the test machine they were fetched from.	Di-rec-tory
inventory	A directory where all ansible inventory files are stored for the given run.	Di-re-tory
logs	A directory where all log files are stored from the run. Logs here consist of teflo runtime logs, ansible logs, etc.	Di-rec-tory
ansi-ble_orchestrator	The directory under logs directory where ansible logs related to orchestrate actionsare stored	Di-rec-tory
ansible_executor	The directory under logs directory where ansible logs related to execute tasks are stored	Di-rec-tory
<sce-naio_filename_withou	The updated scenario descriptor file(s) (created by teflo). This file can be used to pick up where you left off with teflo. You can easily run another task with this given file. It removes the need from starting a whole run over from the beginning.	File
results.yml	The updated scenario descriptor file (created by teflo).	File

Note: **TEFLO_DATA_FOLDER**, **TEFLO_RESULTS_FOLDER**, **TEFLO_WORKSPACE** are TEFLO environmental variables that are made available during a teflo run. They provide the absolute path for the data folder, results folder and workspace respectively

2.4.4 Examples

This page is intended to provide you with detailed examples on how you can use teflo to perform various actions. Some of the actions below may redirect you to a git repository where further detail is given.

Test Setup & Execution

This section provides you with detailed examples on how teflo can perform test setup and execution using common test frameworks. Below you will find sub-sections with examples for installation and execution using commonly used test frameworks. Each framework has an associated repository containing all necessary files to call teflo to demonstrate test setup and execution. Each of the examples demonstrates how teflo consumes the external automated scripts (i.e, ansible roles/playbooks, bash scripts, etc) to install the test frameworks, and how teflo executes example tests and gets the artifacts of the execution.

Note: These frameworks below are just examples on how you can use teflo to run existing automation you may have to install/setup/execute using common test frameworks. Since teflos primary purpose is to conduct "orchestrate" the E2E flow of a multi-product scenario, it has the flexibility to consume any sort of automation to run against your scenarios test machines defined. This allows you to focus on building the automation to setup test frameworks and then just tell teflo how you wish to run it.

Junit

Please reference the example junit example for all details on how you can execute this example with teflo to run a junit example.

Pytest

Please reference the example pytest example for all details on how you can execute this example with teflo to run a pytest example.

Restraint

Please reference the example restraint example for all details on how you can execute this example with teflo to run a restraint example.

2.4.5 Best Practices

Data pass-through

Please visit the following page to understand how you can pass data within teflo tasks.

Scenario Structure

The intended focus of this section is to provide a standard structure for building multi product scenarios. This is just a standard that can be adopted, as a best practice, but is not required for running teflo. Having a solid structure as a foundation will help lead to easier maintenance during the scenarios lifespan. Along with faster turn around for creating new multi product scenarios.

Note: The advantage to a standard structure allows for other users to easily re-run & update a scenario in their environment with minimal effort. It will also help with consistency in teflo's usage, making it easier to understand someone's scenario.

```
template/
├── ansible
├── ansible.cfg
├── teflo.cfg
├── jenkins
│   ├── build
│   │   ├── ansible.cfg
│   │   ├── auth.ini
│   │   ├── build.sh
│   │   ├── hosts
│   │   └── site.yml
│   ├── Jenkinsfile
│   └── job.yml
├── keys
├── Makefile
├── README.rst
├── scenario.yml
├── common_scenario.yml
└── tests
```

The above scenario structure has additional files that are not required for executing teflo. The extra files are used for easily running the scenario from a Jenkins job. Below are two sections which go into more detail regarding each file.

Teflo Files

Based on the standard scenario structure, lets review the directories and files which teflo consumes. For now, the jenkins directory and the Makefile will be ignored as they are related to creating a Jenkins job and not required for executing teflo.

Name	Description	Required
ansible	The ansible directory is where you can define any ansible files (roles, playbooks, etc) that your scenario needs to use. Teflo will load and use these files as stated within your scenario descriptor file.	No
ansible.cfg	The ansible.cfg defines any settings for ansible that you wish to override their default values. It is **highly** recommend for each scenario to define their own file.	No
teflo.cfg	This is a required teflo configuration file. It is recommended to remove the credentials in the file so the credentials are not under source control; however, when running you can either update the credentials or save the credentials in another teflo.	Yes
keys	The keys directory is an optional directory to set ssh keys used for contacting the machines in the scenario.	No
scenario.yml	The scenario.yml is your scenario descriptor file. This file describes your entire E2E multi product scenario.	Yes
common_scen	This is the scenario file used in the include section of the scenario.yml.	No
tests	This is a directory where all the tests that are run during the execution are stored.	No

With this scenario structure you can easily run teflo from the command line or from a Jenkins job. See the following section for more details.

Handing Off A Scenario

After you successfully created a scenario descriptor file that describes an end to end scenario, it should take minimal effort to hand off the scenario for someone else to run. Especially if you followed the previous topic for a baseline for your scenario structure. You need to send all your files in the scenario structure, and tell them to make minor modifications, which are described below:

If the person handing off the scenario is using the same tools for provisioning (i.e. all your machines are provisioned using OpenStack and you hand off to someone else to run the scenario who also plans to provision their resources using OpenStack), it is really simple. You just need to follow the following steps:

1. Tell them to set their credentials for their resource in their teflo.cfg file, using the same name that you used in your scenario descriptor file. A good tip would be to give them your teflo.cfg file with your credentials removed. Also if using the recommend scenario structure, you should be able to tell the user to set their credentials in a separate teflo.cfg file that they refrence with the **TEFLO_SETTINGS** environment variable prior to running teflo.
2. Tell them to update references to the ssh keys that you used for machine access.

If you are handing off to a person that plans to use a different tool for provisioning this can be more complicated. A good tip for this case would be to tell them provision their systems first and inject the driver machine's ssh key into all the machines, and then redefine their systems in teflo as static machines. If this is not an option, the user would have to redefine each of their systems with the correct keys for the provisioning system they plan to use. Please see the provisioning documentation for all options.

2.4.6 Using Localhost

There may be a scenario where you want to run cmds or scripts on the local system instead of the provisioned resources. There are couple options to be able to do this.

Explicit Localhost

The following is an example of a statically defined local machine:

Example

```
---

name: orchestrate example
description: orchestrate example using local host

provision:
  - name: localhost
    groups: local
    ip_address: 127.0.0.1
    ansible_params:
      ansible_connection: local

orchestrate:
  - name: orc_task1
    description: "Print system information."
    orchestrator: ansible
    hosts: localhost
    ansible_playbook:
      name: ansible/system_info.yml
    cleanup:
      name: cleanup_playbook
      description: "Print system information post execution."
      orchestrator: ansible
      hosts: localhost
      ansible_playbook:
        name: ansible/system_info.yml

  - name: orc_task2
    description: "Mock aka fake a kernel update"
    orchestrator: ansible
    hosts: localhost
    ansible_playbook:
      name: ansible/mock_kernel_update.yml
```

When explicitly defined, this host entry is written to the master inventory file and the localhost will be accessible to ALL the Orchestrate and Execute tasks in the scenario.

Implicit Localhost

As of 1.6.0, The use of any other arbitrary hostname will not be supported to infer localhost. It must be *localhost* that is used as a value to *hosts* in the Orchestrate or Execute sections, Teflo will infer that the intended task is to be run on the localhost.

Example

Here an Orchestrate and an Execute task refer to *localhost*, respectively, that are not defined in the provision section.

```
---
provision:
  - name: ci_test_client_b
    groups:
    - client
    - vnc
    ip_address: 192.168.100.51
    ansible_params:
       ansible_private_ssh_key: keys/test_key

orchestrate:
  - name: test_setup_playbook.yml
    description: "running a test setup playbook on localhost"
    orchestrator: ansible
    hosts: localhost

execute:
  - name: test execution
    description: "execute some test script locally"
    hosts: localhost
    executor: runner
    ignore_rc: False
    shell:
      - chdir: /home/user/tests
        command: python test_sample.py --output-results suite_results.xml
        ignore_rc: True
    artifacts:
      - /home/user/tests/suite_results.xml
```

2.4.7 Using Jinja Variable Data

Teflo uses Jinja2 template engine to be able to template variables within a scenario file. Teflo allows template variable data to be set as environmental variables as well as pass variable data via command line.

You can also store the variable data in a file and provide the file path in teflo.cfg

Here is an example scenario file using Jinja to template some variable data:

```
---

name: linchpin_vars_example
description: template example
```

(continues on next page)

(continued from previous page)

```
provision:
  - name: db2_dummy
    provisioner: linchpin-wrapper
    groups: example
    credential: openstack
    resource_group_type: openstack
    resource_definitions:
      - name: {{ name | default('database') }}
        role: os_server
        flavor: {{ flavor | default('m1.small') }}
        image:  rhel-7.5-server-x86_64-released
        count: {{ count | default('1') }}
        keypair: test-keypair
        networks:
          - {{ networks | default('provider_net_ipv6_only') }}
```

The variable data can now be passed in one of three ways.

Raw JSON

You can pass in the data raw as a JSON dictionary

```
teflo run -s scenario.yml -t provision --vars-data '{"flavor": "m2.small", "name": "test
↪"}'
```

```
teflo run -s scenario.yml -t provision --vars-data '{"flavor": "m2.small", "name": "test
↪"}'
--vars-data '{"count": "2"}'
```

Variable File

You can pass in a variable file in yaml format defining the variable data you need. The variable file needs to be placed in the teflo workspace as **var_file.yml** or as yaml files under **vars directory**

User can also set **var_file** as a parameter in the **defaults section of teflo.cfg**. This way user can avoid passing variable data via command line at every run

Following is the precedence of how Teflo looks for variable data:

1. Via command line
2. defaults section of teflo.cfg
3. var_file.yml under the teflo workspace
4. yml files under the directory vars under teflo workspace

Below is an example of the contents of a variable file template_file.yaml.

```
---
flavor: m2.small
networks: provider_net_cci_5
name: test
```

You can pass in the variable file directly

```
teflo run -s scenario.yml -t provision --vars-data template_file.yml --vars-data '{"count
↪": "2"}'
```

If using teflo.cfg this can be set as below. The var_file param can be a path to the variable file or path to the directory where the variable file is stored. If Teflo identifies it a directory then recursively it looks for all files with .yml or .yaml extension within that directory.

```
[defaults]
var_file=~/template_file.yml
```

```
[defaults]
var_file=~/var_dir
```

The above example will look like

```
teflo run -s scenario.yml -t provision
```

Directory with multiple .yml files

You can pass in a directory path containing multiple .yml files. The code will look for files ending with '.yml'

```
teflo run -s scenario.yml -t provision --vars-data ~/files_dir
--vars-data '{"count": "2", "key": "val"}'
```

Nested Variable Usage

Currently teflo supports nested variable using any of above methods

Note: The nested variable can only be string after parsing

For example:

A nested variable can look like below:

1. nested_var: "hello"
2. nested_var: {{ hey }}
3. nested_var: "hello{{ hey }}"

You can

1. **Use multiple layer nested vars**

   ```
   name: {{ hello }}
   hello: {{ world }}
   world: {{ Hey }}
   Hey: "I'm a developer"
   ```

2. **Use multiple nested variables inside one filed**

```
name: "{{ hello }} {{ world }}"
hello: "asd"
world: {{ Hey }}
Hey: "I'm a developer"
```

3. **Use nested variable in a list or dict**

```
name:
    Tom: {{ TomName }}
    Jack: {{ JackName }}
TomName: "Tom Biden"
JackName: "Jack Chen"
adress:
    - {{ street }}
    - {{ city }}
    - {{ state }}
street: "Boston Street"
city: "Boston"
state: "Massachusetts"
```

Note:

TEFLO_DATA_FOLDER , TEFLO_RESULTS_FOLDER and TEFLO_WORKSPACE are TEFLO
environmental variables that are made available during a teflo run, which can be used in scripts and playbooks. They provide the absolute path for teflo's data folder, results folder and workspace respectively

2.4.8 Using Resource Labels

Teflo provides users with the ability to apply labels to each of the resources in the scenario descriptor file (SDF). This can be done by adding a key **labels** to the resources (assets, actions, executes,reports) in the SDF This is an optional feature.

While issuing the Teflo run/validate command a user can provide **–labels or -l** or **–skip-labels or -sl**. Based on the switch provided Teflo will either pick all the resources that belong to that label for a given task OR skip all the resources that belong to the skip-label

Labels allows Teflo to pick desired resources for a task during teflo run and validate. For every task Teflo looks for the resources matching every label provided at the cli. If it does not find any resources for that label, it does not perform that task. If no labels/skip-labels are provided Teflo considers all the resources that belong to a task

If labels are being used in the SDF and while running a teflo run/validate command a label which is not present in the SDF is given, Teflo will raise an error and exit.

Note: –labels and –skip-labels are mutually exclusive. Only one of the either can be used

Providing labels in the SDF

Labels can be provided as comma separated values or as a list in the SDF

```
 ---
 provision:

- name: laptop
  groups: localhost
  ip_address: 127.0.0.1
  ansible_params:
    ansible_connection: local
  labels: abc,pqr

- name: laptop1
  groups: localhost
  ip_address: 127.0.0.1
  ansible_params:
    ansible_connection: local
  labels: abc
```

To run a task using labels

```
teflo run -s scenario.yml --labels prod_a -t provision -w . --log-level info
```

To run a task using skip-labels

```
teflo run -s scenario.yml --skip-labels prod_a -t provision -t orchestrate -w . --log-
↪level info
```

To run a task using more than one labels or skip-labels

You can provide or skip more than one label at a time

```
teflo run -s scenario.yml --labels prod_a --labels prod_b -t provision -w . --log-level␣
↪info

teflo run -s scenario.yml -l prod_a -l prod_b -t provision -w . --log-level info

teflo run -s scenario.yml -skip-labels prod_a -skip-labels prod_b -t provision -w . --
↪log-level info

teflo run -s scenario.yml -sl prod_a -sl prod_b -t provision -w . --log-level info
```

Orchestrate/Execute Tasks with labels:

When running orchestrate and execute tasks if labels are used, Teflo looks for assets from the scenario_graph that match that label. In case if scenario_graph assets do not have any labels or there are assets that dont match teh given labels then all of these assets are taken into consideration.

In the below example, orchestrate task if run with label 'orc' will run only on asset laptop because of the matching label 'orc', even if the orchestrate task has use group name ' hypervisor' which matches both assets laptop and laptop1 hosts.

```
 ---
provision:

- name: laptop
  groups: hypervisor
  ip_address: 127.0.0.1
  ansible_params:
    ansible_connection: local
  labels: 'orc'

- name: laptop1
  groups: hypervisor
  ip_address: 127.0.0.1
  ansible_params:
    ansible_connection: local

orchestrate:

- name: orc1
  orchestrator: ansible
  hosts: hypervisor
  ansible_playbook:
    name: ansible/template_host_list_block_devices.yml
  labels: orc1
```

In the below example if execute task when run with label 'exe1' , then Teflo considers all the assets in the scenario_graph as none of them match the label .It then will only run on asset laptop1 which matches the host name field in the execute block

```
 ---
provision:

- name: laptop
  groups: hypervisor
  ip_address: 127.0.0.1
  ansible_params:
    ansible_connection: local

- name: laptop1
  groups: hypervisor
  ip_address: 127.0.0.1
  ansible_params:
    ansible_connection: local

```

(continues on next page)

(continued from previous page)

```
execute:

- name: exe1
  orchestrator: ansible
  hosts: laptop1
  playbook:
   - name: ansible/template_host_list_block_devices.yml
  labels: exe1
```

Examples

```
---
provision:
    - name: db2_ci_test_client_a
      groups: client
      provisioner: openstack-libcloud
      provider:
        name: openstack
        credential: openstack
        image: rhel-7.5-server-x86_64-released
        flavor: m1.small
        networks:
          - {{ OS_NETWORK }}
        keypair: {{ OS_KEYPAIR }}
      ansible_params:
        ansible_user: cloud-user
        ansible_ssh_private_key_file: keys/{{ OS_KEYPAIR }}
      labels: prod_a

    - name: laptop
      groups: localhost
      ip_address: 127.0.0.1
      ansible_params:
        ansible_connection: local
      labels: abc, prod_b

    - name: laptop1
      groups: localhost
      ip_address: 127.0.0.1
      ansible_params:
        ansible_connection: local
      labels:
        - pqr
        - lmn

orchestrate:
    - name: ansible/install-certs.yml
      description: "install internal certificates"
      orchestrator: ansible
      hosts: client
```

(continues on next page)

(continued from previous page)

```
      ansible_galaxy_options:
        role_file: roles.yml
      labels: prod_a1

    - name: ansible/junit-install.yml
      description: "install junit framework on test clients"
      orchestrator: ansible
      hosts: laptop
      ansible_galaxy_options:
        role_file: roles.yml
      labels: prod_b
```

Example 1

Using the above SDF example to run provision on resources with labels prod_a and prod_b. Here it will provision **db2_ci_test_client_a** and **laptop** assets

```
teflo run -s resource_labels.yml --labels prod_a --labels prod_b -t provision -w . --log-
↪level info
```

Example 2

Using the above SDF example to run provision and orchestrate on resources with labels abc and prod_b. Here it will provision only asset **laptop** and there will be no provision task for label prod_b It will then run orchestrate task **ansible/install-certs.yml** with label prod_b only as there is no orchestrate resource with label abc

```
teflo run -s resource_labels.yml --labels abc --labels prod_b -t provision -t␣
↪orchestrate -w . --log-level info
```

Example 3

Using the above SDF example to skip resources with label prod_a Here teflo will run through all its tasks only on resources which do not match the label prod_a1 So assets **laptop and laptop1** will be provisioned and orchestrate task **ansible/install-certs.yml** will be executed

```
teflo run -s resource_labels.yml --skip-labels prod_a -w . --log-level info
```

Example 4

To run a task with wrong label 'prod_c' which does not exist in the SDF along with a correct label . Here Teflo will throw an error and exit as it does not find the label prod_c

```
teflo run -s resource_labels.yml --labels prod_c -labels prod_a -t provision -w . --log-
↪level info
```

Listing out labels in a SDF

Teflo has a show command with **–list-labels** option which lists out all the labels that have been defined in the SDF

```
$ teflo --help
  Usage: teflo [OPTIONS] COMMAND [ARGS]...

  Teflo - Interoperability Testing Framework

Options:
  -v, --verbose  Add verbosity to the commands.
  --version      Show the version and exit.
  --help         Show this message and exit.

Commands:
  run       Run a scenario configuration.
  show      Show information about the scenario.
  validate  Validate a scenario configuration.
$ teflo show --help
  Usage: teflo show [OPTIONS]

  Show info about the scenario.

Options:
  -s, --scenario   Scenario definition file to be executed.
  --list-labels    List all the labels and associated resources in the SDF
  --help           Show this message and exit.
```

```
$ teflo show -s resource_labels.yml --list-labels

 --------------------------------------------------
 Teflo Framework v1.0.0
 Copyright (C) 2022 Red Hat, Inc.
 --------------------------------------------------
 2020-05-07 01:06:37,235 WARNING Scenario workspace was not set, therefore the workspace␣
↪is automatically assigned to the current working directory. You may experience␣
↪problems if files needed by teflo do not exists in the scenario workspace.
 2020-05-07 01:06:37,260 INFO -----------------------------------------------------------
↪--------------------
 2020-05-07 01:06:37,260 INFO                                 SCENARIO LABELS
 2020-05-07 01:06:37,261 INFO -----------------------------------------------------------
↪--------------------
 2020-05-07 01:06:37,261 INFO PROVISION SECTION
 2020-05-07 01:06:37,261 INFO -----------------------------------------------------------
↪--------------------
 2020-05-07 01:06:37,262 INFO Resource Name        | Labels
 2020-05-07 01:06:37,262 INFO -----------------------------------------------------------
↪--------------------
 2020-05-07 01:06:37,262 INFO laptop               | ['4.5']
 2020-05-07 01:06:37,263 INFO laptop_1             | ['prod_b']
 2020-05-07 01:06:37,263 INFO -----------------------------------------------------------
↪--------------------
 2020-05-07 01:06:37,263 INFO ORCHESTRATE SECTION
```

(continues on next page)

(continued from previous page)

```
2020-05-07 01:06:37,264 INFO -----------------------------------------------------------
↪--------------------
2020-05-07 01:06:37,264 INFO Resource Name        | Labels
2020-05-07 01:06:37,264 INFO -----------------------------------------------------------
↪--------------------
2020-05-07 01:06:37,265 INFO orchestrate_1        | ['prod_a']
2020-05-07 01:06:37,265 INFO -----------------------------------------------------------
↪--------------------
2020-05-07 01:06:37,266 INFO EXECUTE SECTION
2020-05-07 01:06:37,266 INFO -----------------------------------------------------------
↪--------------------
2020-05-07 01:06:37,266 INFO Resource Name        | Labels
2020-05-07 01:06:37,267 INFO -----------------------------------------------------------
↪--------------------
2020-05-07 01:06:37,267 INFO -----------------------------------------------------------
↪--------------------
2020-05-07 01:06:37,267 INFO REPORT SECTION
2020-05-07 01:06:37,268 INFO -----------------------------------------------------------
↪--------------------
2020-05-07 01:06:37,268 INFO Resource Name        | Labels
2020-05-07 01:06:37,268 INFO -----------------------------------------------------------
↪--------------------
```

2.4.9 FAQS

The following are some answers to frequently asked questions about Teflo.

How Do I. . .

Provision

. . . call teflo using static machines?

You need to define the machine as a static machine in the teflo definition file. See Defining Static Machines for details.

. . . run scripts on my local system?

You need to define your local system as a static resource. See The localhost example for details.

. . . run teflo and not delete my machines at the end of the run?

By default when running teflo, you will run all of teflo's tasks; however, you also have the option to pick and choose which tasks you would like to run, and you can specify it with -t or –task. By using this option, you can specify, all tasks, and just not specify cleanup. See Running Teflo for more details.

. . . know whether to use the Linchpin provisioner?

Its recommended new users onbaording with teflo or new scenarios being developed to use the Linchpin provisioner. Its being adopted as the standard provisioning tool and supports a lot more resource providers that can be enabled in teflo. If you have a pre-existing scenario that is not using a teflo native provisioner specific parameter is also a good candidate to migrate over to using the Linchpin provisioner.

If the pre-existing scenarios use teflo native provisioner specific parameters that Linchpin does not support you will need to continue to use those until Lincphin supports the parameter. Linchpin is also python 3 compatible except for Beaker. This support is still not available. We are working with Beaker development to fully support Beaker client on python 3. Any Beaker scenarios using Python 3 should continue to use the teflo bkr-client provisioner. All other providers are supported in Python 3.

. . . install Linchpin to use the Linchpin provisioner?

Refer to the Teflo_Linchpin_Plugin section to install Linchpin and it's dependencies.

. . . know if my current scenarios will work with the new Linchpin provisioner?

You can add the provisioner key to use the linchpin-wrapper and run the validate command

```
teflo validate -s <scenario.yml>
```

This will diplay warnings on which resource parameters may be supported and error out on parameters that are not supported by the provisioner. Resolve any of the warnings and failures. Once validate passes then the scenario should be Linchpin compatible.

. . . parallel provisioning fails with linchpin provisioner ?

Linchpin version 1.9.1.1 introduced issue where when provison concurrency is set to True in teflo.cfg file the provisioning hangs. This can be addressed by setting task concurrency to provision= False in the teflo.cfg. This issue is now fixed with Linchpin version 1.9.2.

. . . which Linchpin version to use ?

Recommended version of linchpin to use is 1.9.2. Lower version will give errors like ModuleNotFoundError: No module named 'ansible.module_utils.common.json' or ansible requirements mismatch or concurrency issues

. . . what versions of python are supported by Linchpin ?

Teflo uses Linchpin to provision openstack, aws, libvirt with python 2 and 3. For beaker Linchpin supports python 3 only with beaker 27 client on Fedora 30 and RH8 systems.

Orchestrate

. . . pass data from playbook to playbook using teflo?

See the Data Pass-through Section

Execute

. . . have my test shell command parsed correctly?

When crafting your complex commands you need to consider a couple items:

- proper YAML syntax
- proper Shell escaping

You can refer to any online YAML validator to make sure the test command is valid YAML syntax. Then you need to remember to make sure you have proper shell escaping syntax to make sure the test command is interpreted properly. Refer to the *Using Shell Parameter for Test Execution* section in the Execute page.

Report

. . . import an artifact that wasn't collected as part of Execute?

You can place the file(s) or folder(s) in the teflo's *<data_folder>/.results* and let teflo search for it or once in the results directory define in it in the *artifact_locations* key telling teflo where to look. Refer to the *Finding the right artifacts* section on the Report page.

. . . stop finding duplicate artifacts during the import?

The driving factor is the name field of the report block. You can narrow and restrict the search based on the shell pattern specified.

For example, if you specify an artifact like *SampleTest.xml* but the artifact has been collected numerous times before its possible a list of the same file in different locations within the teflo *<data_folder>* are going to be found. You can restrict the search to a particular instance by doing something like *test_driver/SampleTest.xml* with test_driver being a directory. Telling teflo to look in that particular directory for the artifact.

Miscellaneous

. . . see the supported teflo_plugins?

See the matrix which calls out all the supported versions for the teflo_plugins for importers and provisioners and related libraries *here*

2.5 Developer's Guide

2.5.1 Architecture Details

Architecture

This page is intended to explain the architecture behind teflo. We strongly recommend that you review the scenario descriptor since it will be helpful when following this document.

Basics

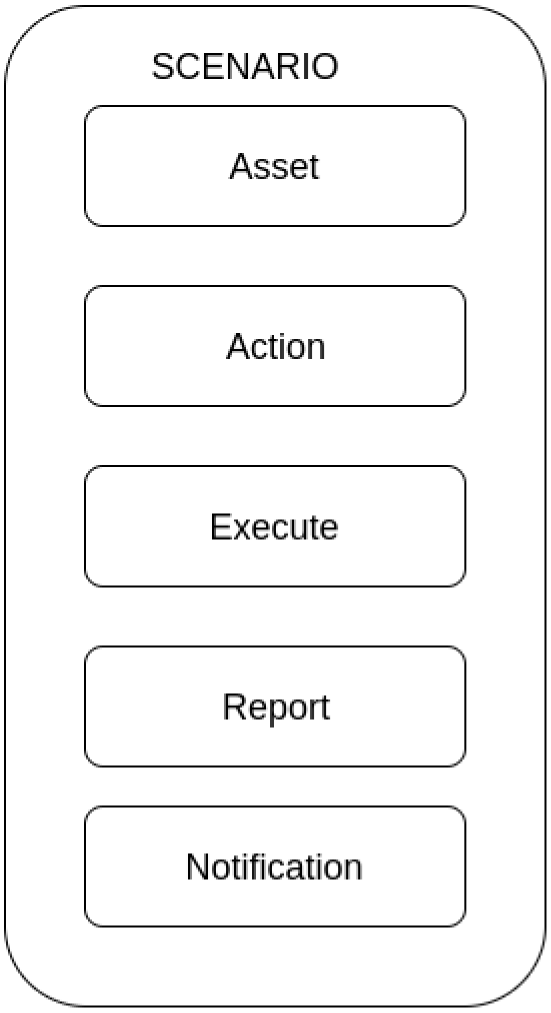
Lets first start with the basics. Teflo is driven by an input file (scenario descriptor). This input file defines the E2E scenario to be processed. Within the file, there are multiple task definitions defined. Each task definition contains resources. These resources will be executed against that given task.

Each teflo execution creates a teflo object for the E2E scenario. The teflo object further creates resource objects for each of the resources provided in the scenario descriptor file. These resources have tasks associated to them to be executed.

Teflo Object

As we just learned from the basics section, the teflo object contains resources. The core resource which makes up the teflo object is a scenario resource. A scenario resource consists of multiple objects of 'resources' which derive tasks for the scenario to be processed. Each resource has a list of associated tasks to it. When teflo executes these tasks the resources associated to that task are used.

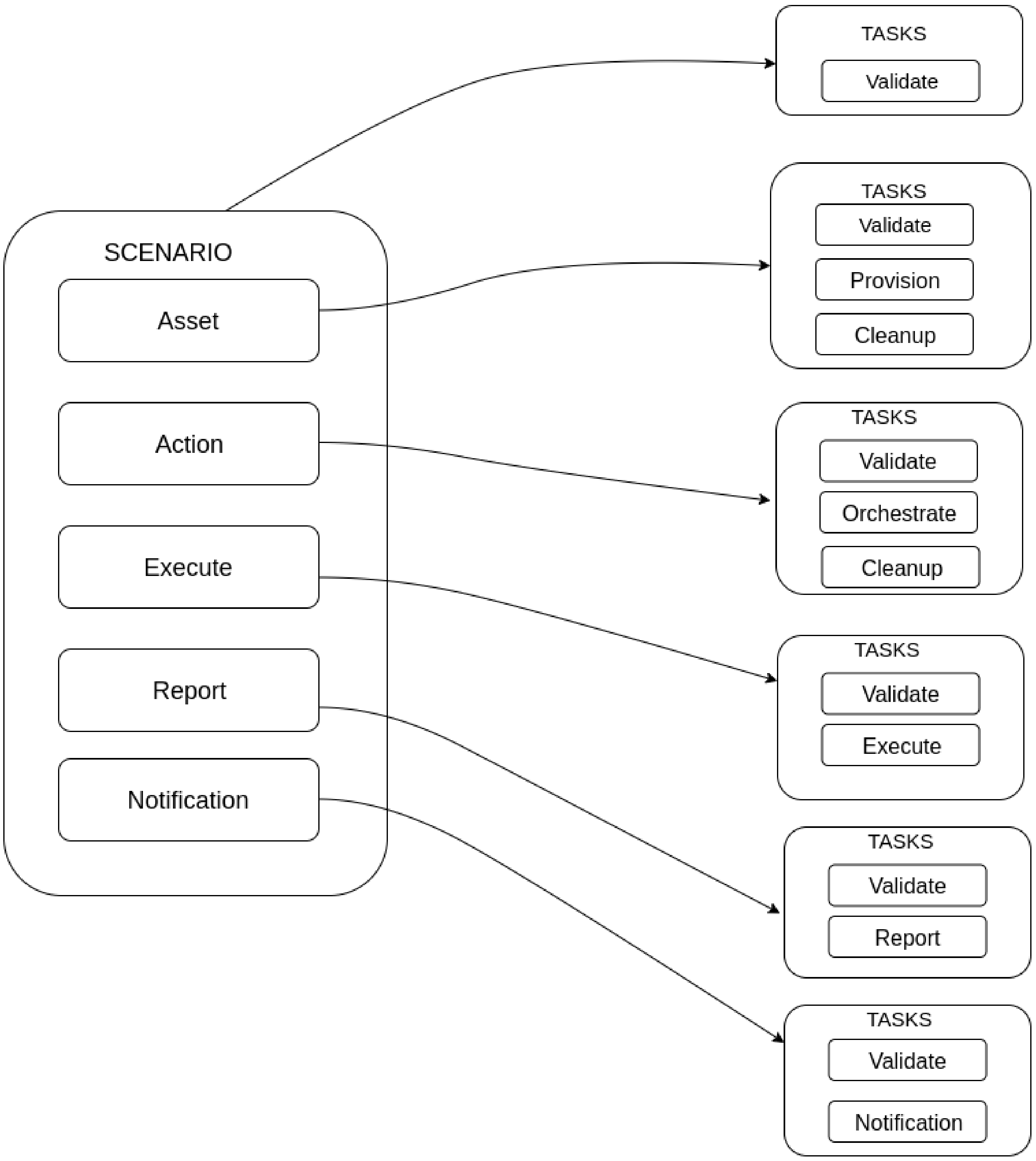
Lets see a diagram with the available resources for a teflo object. The teflo object is made up of scenario resource and the scenario resource comprises of other resources(asset, action, execute, report, notification).



The diagram above shows the resources that make up a scenario resource. The table below describes each resource in more detail.

Re-sourc	Description
Sce-nario	The core resource which makes up the teflo compound. The scenario resource holds the other resources(asset, action, execute and report)
As-set	The asset resources define the system resources for the scenario. These can be anything from hosts, virtual networks, storage, security keys, etc.
Ac-tion	The action resources define the actions to be performed against the defined hosts for the scenario. These actions consist of: system configuration, product installation, product configuration, framework installation and framework configuration. In summary this resource provides the scenario with the ability to perform any remote actions against the hosts defined.
Ex-e-cute	The execute resources define the tests to be executed against the defined host resources.
Re-port	The report resources defines which reporting and analysis system to import test artifacts generated during the execution phase.
No-tifi-ca-tion	The notification resources defines which tool to send the notification to based on the triggers

Now that we have knowledge about how a teflo object is constructed. Which includes a number of resources. Lets dive deeper into the resources. What do we mean by this? Every resource has a number of tasks that it can correspond to.



The diagram above shows the teflo object with resources defined. Each of those resources then have a list of tasks associated to it. This means that when teflo runs a scenario, for each task to be processed it will run the given resources associated to that given task.

e.g. The scenario resource has validate task. This means that when teflo runs the validate task it will process the scenario resource.

e.g. The asset resource has a validate, provision and clean up task. This means that when teflo runs the validate task it will process that asset resource. When it goes to the provision task, it will process that asset resource and the same for clean up task.

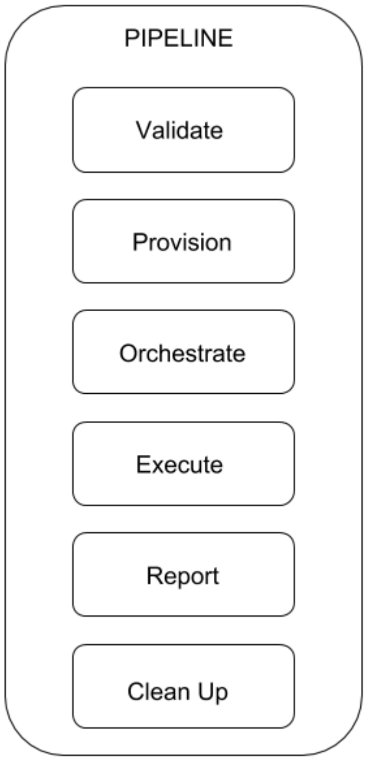
e.g. The action resource has a validate and orchestrate task. This means that when teflo runs the validate task it will process that action resource. When it goes to the orchestrate task, it will process that action resource.

This same logic goes for the execute and report resources.

Teflo Pipeline

In the previous section about teflo object we learned about how a teflo object is constructed with resources and tasks. Every resource could have different tasks. Those tasks are executed in a certain order which the user can provide.

Lets see a diagram showing the default tasks that will get executed when running teflo.



The above diagram shows the ordered list from top to bottom of the tasks teflo will execute.

If no resources are associated to a given task, teflo would skip executing the task. This provides the user with the ability to control the flow of their scenario.

Plug And Play

Teflo was developed with the OO programming model which allows it to be easily extended. This means teflo is very flexible at being able to interface with various applications. Teflo supports a plugin model where it can interface with different plugins created for teflo. Teflo has interfaces for provisioners, orchestartors, executors, importers and notifiers. These interfaces allow differnt plugins to work with teflo.

The best way to explain this is to go through a couple examples. First we will look at how this relates to asset resources and the provision task.

Every asset resource defined within a teflo object has an associated provisioner to it. This allows the user to select different tools to handle the provision task request. Teflo provides an asset_provisioner interface which can talk to differnt provisioners, e.g. bkr_client_plugin, os_libcloud_plugin, etc.

```
teflo/provisioners
├── asset_provisioner.py
├── ext
│   ├── bkr_client_plugin
│   │   ├── beaker_client_plugin.py
│   │   ├── __init__.py
│   │   └── schema.yml
│   ├── __init__.py
│   └── os_libcloud_plugin
│       ├── __init__.py
│       ├── openstack_libcloud_plugin.py
│       └── schema.yml
├── __init__.py
```

```yaml
---
name: demo
description: demo

provision:
  - name: ccit_ci_test_client_a
    groups: client, test_driver
    provisioner: openstack_libcloud
    credential: openstack
    key_pair: ccit_key
    image: rhel-7.4-secommonrver-x86_64-released
    flavor: m1.small
    network:
     - private_network
     - provider_net_cci_8
    ansible_params:
      ansible_user: cloud-user
      ansible_ssh_private_key_file: keys/ccit_key
```

The above code snippets demonstrate how from the asset resource definition defined within the scenario descriptor file. It tells teflo that it would like it to use the openstack_libcloud provisioner. With this flexibility users could provide their own module to provision and define this as the provisioner for their given asset resource.

Teflo uses bkr_client_plugin(using beaker client) and os_libcloud_plugin (using openstack libcloud) as its native provisioner plugins. The implementation for users to plug in their own provisioner can be possible by creating a separate provisoner plugin. We currently have external provisioner plugins for linchpin and openstack-client

Here is an example based on a custom provisioner module:

```
teflo/provisioners
├── beaker.py
├── ext
│   └── __init__.py
├── __init__.py
├── openshift.py
├── openstack.py
└── provisioner_xyz.py
```

```yaml
---
name: demo
description: demo

provision:
    - name: machine1
      provisioner: provisioner_xyz        # provisioner name
      credential: openstack-creds
      image: image1
      flavor: flavor
      networks:
        - network
      floating_ip_pool: 0.0.0.0
      keypair: keypair
      groups: group1
```

Note: Please visit Developers Guide to understand more about how to create a customized plugin for Teflo

Plugin model also applies to the other resources within the teflo object. Lets look at the action resource. Teflo provides a orchestrator interface called action_orchestrator which will interface with different orchestrator plugins. This resources main purpose is to perform configuration actions. To do configuration there are a lot of tools that currently exists to perform these actions. By default teflo supports the ansible orchestrator plugin out of the box. It can easily be plugged in to use a different orchestrator.

Here is an example with an action resource using the default ansible orchestrator by teflo.

```
teflo/orchestrators/
├── _ansible.py
├── _chef.py
├── ext
│   └── __init__.py
├── __init__.py
└── _puppet.py
```

```yaml
---
name: demo
description: demo

provision:
  - name: ccit_ci_test_client_a
    groups: client, test_driver
```

(continues on next page)

(continued from previous page)

```
    provisioner: openstack_libcloud
    credential: openstack
    key_pair: ccit_key
    image: rhel-7.4-secommonrver-x86_64-released
    flavor: m1.small
    network:
     - private_network
     - provider_net_cci_8
    ansible_params:
      ansible_user: cloud-user
      ansible_ssh_private_key_file: keys/ccit_key

orchestrate:
    - name: rhn_subscribe
      orchestrator: ansible         # orchestrator name
      hosts:
        - machine1
      vars:
        rhn_hostname: <hostname>
        rhn_user: <user>
        rhn_password: <password>
```

It can easily be extended to work with other various orchestrators.

Conclusion

Hopefully after reading this document you were able to have a better understanding on how teflo was designed. To gain an even deeper understanding on how it works. We highly recommend following the development document to step through the code.

2.5.2 Development Information

Development

Welcome!

The teflo development team welcomes your contributions to the project. Please use this document as a guide to working on proposed changes to teflo. We ask that you read through this document to ensure you understand our development model and best practices before submitting changes.

Any questions regarding this guide, or in general? Please feel free to file an issue

Release Cadence

The release cadence of the project follows the rules below, all contributions are welcome and please be aware of the cadence. For general usage users/contributors can fork the develop branch in order to use the latest changes

1. **Develop** is the branch contributions are made on
2. **Master** branch is the stable branch
3. **Release a new version in every 5 weeks**
    1. Changes will be evaluated and merged into master ,if suitable ,from develop branch and then released to the PyPi server
4. **The release will be major/minor/patch based on :**
    1. Major changes, e.g. major refactor or backward compatibility break, etc. (major release)
    2. Other changes like new features or code refactoring that are not major (minor)
    3. Bug fixes (patch)
5. **Labels are recommended for issues and PRs in the following manner**
    1. **Critical** : For any urgent blocking issues
    2. **Bug** : For any bugs
    3. **New_feature** : For any new feature request
6. **Hotfix release may be available before usual release cycle based on issue severity. A hotfix release is considered if:**
    1. It is blocking user automation and no workaround is available
    2. Develop branch installation does not unblock the user.

Branch Model

Teflo has two branches

- **develop** - all work is done here
- **master** - stable tagged release that users can use

The master branch is a protected branch. We do not allow commits directly to it. Master branch contains the latest stable release. The develop branch is where all active development takes place for the next upcoming release. All contributions are made to the develop branch.

Most contributors create a new branch based off of develop to create their changes.

How to setup your dev environment

Lets first clone the source code. We will clone from the develop branch.

```
$ git clone https://github.com/RedHatQE/teflo.git -b develop
```

Next lets create a Python virtual environment for teflo. This assumes you have virtualenv package installed.

```
$ mkdir ~/.virtualenvs
$ virtualenv ~/.virtualenvs/teflo
$ source ~/.virtualenvs/teflo/bin/activate
```

Now that we have our virtual environment created. Lets go ahead and install the Python packages used for development.

```
(teflo) $ pip install -r teflo/test-requirements.txt
```

Let's create our new branch from develop. Do this step from teflo's root folder

```
(teflo) $ cd teflo
(teflo) $ git checkout -b <new branch>
(teflo) $ cd ..
```

Finally install the teflo package itself using editable mode.

```
(teflo) $ pip install -e teflo/.
```

You can verify teflo is installed by running the following commands.

```
(teflo) $ teflo
(teflo) $ teflo --version
```

You can now make changes/do feature development in this branch

How to run tests locally

We have the following standards and guidelines

- All tests must pass
- Code coverage must be above 50%
- Code meets PEP8 standards

Before any change is proposed to teflo we ask that you run the tests to verify the above standards. If you forget to run the tests, we have a github actions job that runs through these on any changes. This allows us to make sure each patch meets the standards.

We also highly encourage developers to be looking to provide more tests or enhance existing tests for fixes or new features they maybe submitting. If there is a reason that the changes don't have any accompanying tests we should be annotating the code changes with TODO comments with the following information:

- State that the code needs tests coverage
- Quick statement of why it couldn't be added.

```
#TODO: This needs test coverage. No mock fixture for the Teflo Orchestrator to test with.
```

How to run unit tests

You can run the unit tests and verify pep8 by the following command:

```
(teflo) $ make test-functional
```

This make target is actually executing the following tox environments:

```
(teflo) $ tox -e py3-unit
```

Note: we use a generic tox python 3 environment to be flexible towards developer environments that might be using different versions of python 3. Note the minimum supported version of python is python 3.6.

How to run localhost scenario tests

The local scenario test verify your changes don't impact core functionality in the framework during provision, orchestrate, execute, or report. It runs a scenario descriptor file using localhost, a teflo.cfg, some dummy ansible playbooks/scripts, and dummy test artifacts. It does NOT run integration to real external system like OpenStack or Polarion.

```
(teflo) $ make test-scenario
```

This make target is actually executing the following tox environments:

```
(teflo) $ tox -e py3-scenario
```

Note: If there is a need to test an integration with a real external system like OpenStack or Polarion, you could use this scenario as a basis of a more thorough integration test of your changes. It would require modifying the scenario descriptor and teflo.cfg file with the necessary parameters and information. But it is not recommended to check in this modified scenario as part of your patch set.

How to propose a new change

The teflo project resides in Red Hat QE github space. To send the new changes you will need to create a PR against the develop branch . Once the PR is sent, the github actions will runt the unit tests and will inform the maintainers to review the PR.

At this point you have your local development environment setup. You made some code changes, ran through the unit tests and pep8 validation. Before you submit your changes you should check a few things

If the develop branch has changed since you last pulled it down. it is important that you get the latest changes in your branch. You can do that in two ways:

Rebase using the local develop branch

```
(teflo) $ git checkout develop
(teflo) $ git pull origin develop
(teflo) $ git checkout <branch>
(teflo) $ git rebase develop
```

Rebase using the remote develop branch

```
(teflo) $ git pull --rebase origin/develop
```

Finally, if you have mutiple commits its best to squash them into a single commit. The interactive rebase menu will appear and guide you with what you need to do.

```
(teflo) $ git rebase -i HEAD~<the number of commits to latest develop commit>
```

Once you've completed the above you're good to go! All that is left is to submit your changes to your branch and create a new PR against the develop branch

Submitting the PR

Once a set of commits for the feature have been completed and tested. It is time to submit a Pull Request. Please see the github article to get an idea about submitting a PR, Creating a pull request.

Guidelines for submitting the PR

1. Submit the Pull Request (PR) against the **develop** branch.
2. Provide a ticket number if available in the title
3. Provide a description.

Once the PR is created, it will need to be reviewed, and CI automation testing must be executed. It is possible that additional commits will be needed to pass the tests, address issues in the PR, etc.

Once the PR is approved, it can be merged.

You can also install the github cli and send PRs using gh cli More information on how to install and where to find binaries is here

When using the cli first time from your terminal you may have to authenticate your device. If web option is used it opens up a browser to put in the given code

```
$ gh auth  login --web
- Logging into github.com

! First copy your one-time code: ABCD-ABCD
- Press Enter to open github.com in your browser...
This tool has been deprecated, use 'gio open' instead.
See 'gio help open' for more info.

✓ Authentication complete. Press Enter to continue...

✓ Logged in as user123
```

Once you are authenticated you can send in the PR, using the create command, It will ask certain questions and then ask you to submit the PR.

More information on how to use gh cli

```
$ gh pr create --title "Feature umb importer" --reviewer rujutashinde --base develop
 Warning: 9 uncommitted changes
 ? Where should we push the 'tkt_218' branch? Skip pushing the branch

```

(continues on next page)

(continued from previous page)

```
Creating pull request for tkt_218 into develop in RedHatQE/teflo

? Body <Received>
? What's next? Submit
https://github.com/RedHatQE/teflo/pull/01
```

Note: Merging is currently done only by the maintainers of the repo This will be opened up to contributors at a future time

Feature Toggles

Although this doesn't happen very often this does warrant a mention. If a feature is too big to, where it would better suited to merge incrementally in a 'trunk' style of development. Then we should consider utilizing feature toggles so as the develop branch can stay releasable at all times.

The teflo.cfg is capable of reading feature toggles and utilizing them. It's a very rudimentary implementation of a feature toggle mechanism but it has worked in the past on short notice. Below is the process when working at adding functionality to one of the main resources (Host, Actions, Executions, Reports).

To the resource we are working on define the following feature toggle method

```
def __set_feature_toggles_(self):

self._feature_toggles = None

for item in self.config['TOGGLES']:
    if item['name'] == '<name of resource the feature belongs to>':
        self._feature_toggles = item
```

Then in the __init__ function of the resource you are working on add the following lines of code. This will help to keep teflo running original code path unless explicitly told to use the new feature

```
if self._feature_toggles is not None and self._feature_toggles['<name of new feature␣
↪toggle>'] == 'True':
    <new feature path>
else:
    <original code path>
```

Now in your teflo config file when you want to use the new code path for testing or continued development you can do the following:

```
[orchestrator:ansible]
log_remove=False
verbosity=v

[feature_toggle:<resource name from step 1>]
<feature toggle name specified in step 2>=True
```

How to build documentation

If you are working on documentation changes, you probably will want to build the documentation locally. This way you can verify your change looks good. You can build the docs locally by running the following command:

```
(teflo) $ make docs
```

This make target is actually executing the following tox environments:

```
(teflo) $ tox -e docs
```

How to write an plugin for teflo

For developers who wish to put together their own plugins can use Teflo's plugin templates to do so. The plugin templates creates a directory with required imports from teflo project based on the plugin type to be created (provisioner/orchestrator/executor/importer/notification). Once templates are in place developers can then go ahead with actual plugin work

How to use plugin templates

To use this template to create your plugin folder:

1. install cookiecutter

```
pip install cookiecutter
```

2. Clone the teflo_examples repo

```
git clone git@github.com:RedHatQE/teflo_examples.git
```

3. Go to the space where you want your plugin folder to be created then run the command

```
cookiecutter <path to the cloned teflo_examples repo>/teflo_plugin_template
```

4. When you run this you will be prompted to provide values for the variables in the cookiecutter json file, Below are the variables and their description. User should provide the values it needs, else the default values will be taken

Variable	Description	Default Value
teflo_plugin_t	type of teflo plugin to be created (provisioner or orchestrator or executor or importer or notification)	provisioner
direc-tory_name	name to be give to the plugin repo directory.	teflo_provisionerX_plugin
plu-gin_name	name of the python file where your actual plugin code will reside	provx_plugin
plu-gin_class_nam	the name of the class within the python file	ProvXProvisionerPlugin
test_class_nan	name to be given to the unit test file under tests folder. This is auto generated if left blank	test_provx_plugin
plu-gin_descriptio	Plugins description that goes into the setup.py	teflo provisioner plugin
jenk-ins_ci_job_lin	jenkins ci job link once you have created that. This gets updated in the jenkins/Jenkinsfile	your ci job link
plugin_url	plugin url needed to start the ci job. This gets updated in the jenk-ins/Jenkinsfile	plugin url on gitlab/github
authors	The value that gets updated in the AUTHORS file	CCIT tools dev team <ci-ops-qe@redhat.com>

Note: Here the variables **jenkins_ci_job_link** and **plugin_url** can be left default, and then these values can be updated in the jenkins/Jenkinsfile once user has the CI job url and repo url ready. These variables are meant to be more as a place holder for users to know where they can update later

Note: Read here about cookiecutter package

Example

Example to use the plugin template

Template Guidelines

Note: The above plugin template repo was created based on the following guidelines. These are meant for developers to understand. It is recommended for developers to make use of the template while working on Teflo Plugins

1. The new plugin will need to import one of these Teflo classes based on the plugin they wish to develop Teflo Plugin classes: **ProvisionerPlugin OrchestratorPlugin ExecutorPlugin ImporterPlugin NotificationPlugin** from the **teflo.core** module.
2. It should have the plugin name using variable **__plugin_name__**
3. **It should implement the following key functions**
    - For provisioner plugins implement the **create**, **delete**, and **validate** functions
    - For importer plugins implement the **import_artifacts** and **validate** functions

4. You should define a schema for Teflo to validate the required parameter inputs defined in the scenario file. Teflo use's pyqwalify to validate schema. Below is an example schema

```yaml
---
# default openstack libcloud schema

type: map
allowempty: True
mapping:
  image:
    required: True
    type: str
  flavor:
    required: True
    type: str
  networks:
    required: True
    type: seq
    sequence:
      - type: str
  floating_ip_pool:
    required: False
    type: str
  keypair:
    required: False
    type: str
  credential:
    required: False
    type: map
    mapping:
      auth_url:
        type: str
        required: True
      username:
        type: str
        required: True
      password:
        type: str
        required: True
      tenant_name:
        type: str
        required: True
      domain_name:
        type: str
        required: False
      region:
        type: str
        required: False
```

Once you've created your schema and/or extension files. You can define them in the plugin as the following attributes **__schema_file_path__** and **__schema_ext_path__**.

```python
__schema_file_path__ = os.path.abspath(os.path.join(os.path.dirname(__file__),
```

(continues on next page)

(continued from previous page)

```
                                                    "files/schema.yml"))
__schema_ext_path__ = os.path.abspath(os.path.join(os.path.dirname(__file__),
                                                   "files/lp_schema_extensions.py"))
```

To validate the schema, you can import the **schema_validator** function from the **teflo.helpers** class

```
# validate teflo plugin schema first
    schema_validator(schema_data=self.build_profile(self.host),
                     schema_files=[self.__schema_file_path__],
                     schema_ext_files=[self.__schema_ext_path__])
```

5. To enable logging you can create a logger using the **create_logger** function or calling python's **getLogger**
6. The plugin needs to add an entry point in its setup.py file so that it can register the plugin where Teflo can find it. For provsioners register the plugin to **provisioner_plugins** and for importers register to **importer_plugins**. Refer the example below:

```
from setuptools import setup, find_packages

setup(
    name='new_plugin',
    version="1.0",
    description="new plugin for teflo",
    author="Red Hat Inc",
    packages=find_packages(),
    include_package_data=True,
    python_requires=">=3",
    install_requires=[
        'teflo@git+https://code.engineering.redhat.com/gerrit/p/teflo.git@master',
    ],
    entry_points={
                  'importer_plugins': 'new_plugin_importer = <plugin pckage name>
↪:NewPluginClass'
                 }
)
```

Please refer here for more information on entry points

Example for plugin:

```
from teflo.core import ImporterPlugin

class NewPlugin(ImporterPlugin):

    __plugin_name__ = 'newplugin'

    def __init__(self, profile):

        super(NewPlugin, self).__init__(profile)
        # creating logger for this plugin to get added to teflo's loggers
        self.create_logger(name='newplugin', data_folder=<data folder name>)
        # OR
        logger = logging.getLogger('teflo')
```

(continues on next page)

(continued from previous page)

```
    def import_artifacts(self):
        # Your code
```

2.6 Glossary

Ansible
Open source software that automates software provisioning, configuration management, and application deployment. Ansible

beaker
Resource management and automated testing environment.

Cachet
An open source status page system. Cachet

teflo
Test Execution Framework Libraries and Onjects

credentials (section)
Required credential definitions for each resource that needs to be provisioned. Credentials are set in teflo.cfg file. They are referenced by name in the scenario.

dnf
A software package manager that installs, updates, and removes packages on RPM-based Linux distributions.

E2E
end to end

execute (section)
Defines the location, setup, execution and collection of results and artifacts of tests to be executed for a scenario.

git
A version-control system for tracking changes in computer files and coordinating work on those files among multiple people.

Jenkins
Open source automation server, Jenkins provides hundreds of plugins to support building, deploying and automating any project. Jenkins

orchestrate (section)
Defines the configuration and setup to be performed on the resources of a scenario in order to test the system properly.

pip
A package management system used to install and manage software packages written in Python.

provision (section)
Defines a list of resources and there inputs to be provisioned.

PyPI
The Python Package Index (PyPI) is a repository of software for the Python programming language.

report (section)
Defines the reporting mechanism of a scenario.

resource (teflo)
A host/node to provision or take action on.

resource (external)
External components that a scenario requires to run.

resource check (section)
Specifies a list of external resource components to check status of before running scenario. If any component not available the scenario will not run.

role (ansible)
Ways of automatically loading certain vars_files, tasks, and handlers based on a known file structure. Grouping content by roles allows easy sharing of roles with other users.

groups (teflo)
The function assumed or part played by a node/host. Specified in provision section.

section (teflo)
The major areas a scenario is broken into. Sections of a scenario relate to a particular component within teflo. Valid sections are 'Resource Check', Credentials, Provision, Orchestrate, Execute and Report.

scenario (teflo)
A teflo scenario descriptor file. Teflo input file. SDF.

SDF
scenario descriptor file

task
An action to be performed.

task (ansible)
A call to an ansible module.

task (teflo)
Actions that are run against a scenario. Valid tasks are validate, provision orchestrate, execute, report and cleanup.

task (orchestrate)
A configuration action that will then correlate to an orchestrators task. The default orchestrator for teflo is Ansible.

tox
A generic virtualenv management and test command line tool.

virtualenv
A tool to create isolated Python environments. Virtualenv

YAML
A human-readable data serialization language. It is commonly used for configuration files, but could be used in many applications where data is being stored or transmitted.

yum
Yellowdog Updater, Modified (YUM) is an open-source command-line package-management utility for computers running the GNU/Linux operating system using the RPM Package Manager

2.7 Changelog

Version 2.4.0 (2023-04-05) This will be the last official release! Bug Fixes and Documentation Modification ~~~~~~~~~ * Fixed ssh-python issue * fix to get collection playbook installed in default path * Correction in Rtd version and doc change

2.7.1 Maintenance

- Removed unsupported containers from github actions
- Raise error when provider key is used
- added reduced pre-commit hooks

Version 2.3.0 (2023-02-06)

2.7.2 Bug Fixes

- Changed min python to v3.9 and ansible version to 2.14.0
- fix to get collection playbook installed in default path

Version 2.2.9 (2022-11-15)

2.7.3 Bug Fixes

- Fix install rsync on different versions

Version 2.2.8 (2022-11-7)

2.7.4 Enhancements

- Added Support of Teflo aliases.

2.7.5 Bug Fixes

- Fix invalid value error for command show –show-graph -im.
- Added repo install if failed to find rsync package.
- Fix allow roles to be installed correctly from req files with only a list.

Version 2.2.7 (2022-09-19)

2.7.6 Documentation

- update Teflo and Teflo plugins copyright to 2022.

2.7.7 Enhancements

- Improvements to downloading ansible roles/collections
- Upgrade urllib3

2.7.8 Bug Fixes

- Beaker Provisioner: append to authorized_keys rather than overwrite it
- Fix on_start trigger

Version 2.2.6 (2022-07-25)

2.7.9 Documentation

- Added comments in the pipeline.py to clarify the usage of filters.

2.7.10 Enhancements

- Added strict validation to bkr_client schema.
- Added support of git ssh to clone remotes.
- Added coverage xml file in Unittest

2.7.11 Bug Fixes

- Fix running ansible collection

Version 2.2.5 (2022-05-16)

2.7.12 Enhancements

- Added support of ansible group_vars files.
- Add support to grab ipv4 when node has multiple addresses.
- Add unit tests for notification problem

Version 2.2.4 (2022-04-18)

2.7.13 Bug Fixes

- Fixed the jinja template issue
- Upgrade Sphinx to be compatible with jinja2 v3.1.1
- Silence notify messages when no notifications enabled
- Fixed for Teflo does not take into account provision resources that do not match the supplied teflo label
- Fixed for DISPLAY_SKIPPED_HOSTS option is deprecated

Version 2.2.3 (2022-03-11)

2.7.14 Bug Fixes

- Fixed issue with centos 8 image for unit tests
- Fixed ansible warnings in stderr
- Fixed preserve whitespace when dumping ansible output

Version 2.2.2 (2022-01-31)

2.7.15 Enhancements

- make scenario graph size a static attribute
- Allow ANSIBLE_EXTRA_VARS_FILES option for orchestrate/execute task to pick up variable files provided via cli

2.7.16 Bug Fixes

- Allow IPv6 addresses SSH connection validation
- Fixed nested var issue
- Fixed duplicate resource name issue

Version 2.2.0 (2021-12-11)

2.7.17 Features

- From this release, users are able to define remote_workspace in sdf file and use remote scenario

2.7.18 Enhancements

- Make env variables available during Orchestrate and execute stage of Teflo run
- Added __hash__ and __eq__ for Teflo Resource class

2.7.19 Bug Fixes

- Fixed notification to display passed and failed tasks for the entire scenario_graph
- Fixed "for running You have to provide a valid scenario file. fails with 'skip-fail' KeyError"

Version 2.1.0 (2021-11-05)

2.7.20 Documentation

- Modified quickstart page and flowchart for teflo

2.7.21 Enhancements

- Make the data folder and results folder available to users in the form of environment variables
- Added support usage of variables in the variables files in message notification templating
- Add skip failures ability during the graph run
- Allow iterate_method from cli
- Added check for installing ansible roles when running ansible playbooks under resource_check method

2.7.22 Bug Fixes

- Fixed syntax warnings in CI
- Fix same file error
- Fixed test result summary does not take into account error test case elements
- Fixed the ansible nested var issue
- Fix issues of jinja templating in include

Version 2.0.0 (2021-08-02)

2.7.23 Features

- Recursive include of child scenarios is supported with scenario graph implementation
- Replaced scenario_streams with the newly added scenario graph
- teflo show -s sdf_file.yml –show-graph added, users can see the whole scenario graph structure
- Added term color to display log messages red(for errors) and green for other information
- Added support for selecting the scenario execution order __by_level__ and __by_depth__ using the *included_sdf_iterate_method* parameter in teflo.cfg

2.7.24 Enhancements

- Redesigned teflo execution pipeline
- Redesigned the cleanup logic for scenarios
- Redesigned the validate logic for scenarios
- Redesigned the results generation
- Redesigned the inventory generation(output inventory stays the same, the logic behind the scene changed)
- Added typing for many functions(e.x *def func(param:list=[]):->str*)
- Added tostring,path,pullpath,inventory methods to scenario class

2.7.25 Documentation

- Added explanation about how to use scenario graph
- Added explanation about how *include* works with scenario graph

Version 1.2.5 (2021-11-05)

2.7.26 Enhancements

- Enabled ci for version 1.2.x

2.7.27 Bug Fixes

- Fix for: custom resource_check does not honor the ansible_galaxy_options
- Fixed the ansible nested var issue with ansible_facts

Version 1.2.4 (2021-09-23)

2.7.28 Enhancements

- beaker provisioner total attempts to an integer data type
- add space to beaker warning
- Allow users to set ansible verbosity using ansible environment variable

2.7.29 Bug Fixes

- invalid inventory generated when groups contains the machine name
- Report task fails when executes attribute is used and No asset is present

Version 1.2.3 (2021-08-02)

2.7.30 Features

- Add the var-file declared by user as an extra_vars in the ansible orchestrate and execute task
- teflo_rppreproc_plugin to support RPV5 instances

2.7.31 Enhancements

- support –vars-data w/show command
- Added support bkr’s ks-append(s) option in beaker-client plugin

2.7.32 Bug Fixes

- Added a generic exception handling during ssh to hosts
- Added fix for resource ordering issue in results.yml
- update import_results list when is not None
- Using variable files with variables as list/dict causes an exception

2.7.33 Documentation

- Correction in documentation to point to fixed gh_pages
- Added release cadence to Contribution.rst
- Added workaround(use of shell script) to allow make docs-wiki work correctly using makefile

Version 1.2.2 (2021-07-16)

2.7.34 Features

- Added teflo init command (It will generate a genralized teflo workspace for you with examples)
- Added openstack instance metadata field for os_libcloud_plugin

Version 1.2.1 (2021-06-28)

2.7.35 Features

- Introduced teflo_notify_service_plugin, users can use this plugin to send out messages to many platforms now

2.7.36 Enhancements

- Added new default location for the usage of variables, you can now put varfile in default locations without specifying the with –vars-data
- Added nested recursive variable support, now the users can use variable inside a variable in your variable file
- Added ability to pass multiple files to the extra_vars module
- Create root users ssh directory for beaker provisioner when non existing
- Added teflo_notify_service_plugin, terraform-plugin and webhook-notification-plugin to setup.py extra require, users can do something like 'pip install teflo[teflo_notify_service_plugin]' now

2.7.37 Bug Fixes

- Fixed Ansible version bug

2.7.38 Documentation

- Updated compatibility matrix
- Updated some installation guide for some plugins
- Update teflos package classifiers

Version 1.2.0 (2021-05-10)

2.7.39 Features

- Introduced teflo_terraform_plugin, users can use terraform during provision phase now

2.7.40 Enhancements

- Use pyssh over paramiko library

2.7.41 Bug Fixes

- Hosts are not correctly resolved when groups are mentioned in the orchestrate task
- Change the copyright license to 2021
- Fix the ansible stderr issue

2.7.42 Documentation

- Modified compatibility matrix
- removed jenkins folder
- Added example in execute.rst

Version 1.1.0 (2021-03-29)

2.7.43 Enhancements

- Improved error messaging for syntax errors in SDF
- Allow jinja templating within teflo.cfg
- Allow multiple –vars-data arguments
- Removed backward compatibility support for using name field under orchestrate block as script/playbook path
- Removed backward compatibility support for using ansible_script as a boolean
- Removed backward compatibility support to remove role attribute from assets, and use only groups

2.7.44 Bug Fixes

- Modified ansible-base version in setup.py
- Fixed issue during generation inentory for static host with no groups attribute
- Fixed issue where Teflo was improperly exiting with a return code of 0 when the scenario descriptor file was invalid

2.7.45 Documentation

- Added more details and diagram on the teflo readme page
- Corrected the vars-data info page
- Use github pages for teflo plugins

Version 1.0.1 (2021-02-10)

2.7.46 Enhancements

- Update teflo config code to not make defaults section mandatory
- For Openstack, display instance IDs
- Alter error message to not contain the words “fail” and “success” simultaneously
- The openstack lincloud schema needs two additional keys project_id and project_domain_id

2.7.47 Bug Fixes

- asset delete fails when using native provisioner (os libcloud) without provider attribute

2.7.48 Documentation

- Updated provision and examples docs to remove provider key and update examples
- Updated contribution page to add plugin template info

Version 1.0.0 (2021-01-07)

This is the first version of Teflo project (formerly known as Carbon)

2.8 Contacts

2.8.1 The Teflo Tool

Framework Community

Information on framework updates can be found on the carbon-watchers list. If you have any questions, need help developing your scenarios, or how best to use teflo to fit your use case, we encourage you to reach out to the carbon-watchers list as we have a diverse community of users that can offer insight on some of these questions.

Please subscribe here.

(WIP) Logging Issues with Teflo

Please log the issue on github here

Note: This section is still WIP, this space will be updated with more info on what additional info can be included in the github issue for correct tracking.

2.8.2 Authors and Maintainers

Please feel free to reach out to any of the maintainers, if you have any questions.

Maintainers

```
Rujuta Shinde <rushinde@redhat.com>
Junqi Zhang <junqzhan@redhat.com>
```

Creators

```
Ryan Williams <rywillia@redhat.com>
Stephen Matula <smatula@redhat.com>
Tiago M. Vieira <tmoreira@redhat.com>
Vimal Patel <vipatel@redhat.com>
```

INDEX