khaleesi-settings documentation

Release 0.1

RDO CI team

Jul 26, 2017

Contents

Contents:

CHAPTER 1

Using Khaleesi

Khaleesi is an ansible based deployment tool for OpenStack. It was developed by the Red Hat Openstack CI team and is used to simplify automation and builds. In order to work, khaleesi needs a configuration file which can either be provided by khaleesi-settings project. Khaleesi-settings populates the config file using the ksgen (Khaleesi settings generator) tool, located in khaleesi project.

> https://github.com/redhat-openstack/khaleesi-settings or http://<redhat-internal-git-server>/git/khaleesi-settings.git

Prerequisites

Fedora21+ with Python 2.7. For running jobs, khaleesi requires a dedicated RHEL7 or F21 Jenkins slave. We do have an ansible playbook that sets up a slave, see *Creating a Jenkins slave*.

> **Warning:** Do not use the root user, as these instructions assumes that you are a normal user and uses venv. Being root may shadow some of the errors you may make like forgetting to source venv and pip install ansible.

Update your system, install git and reboot:

```
sudo yum -y update && sudo yum -y install git && sudo reboot
```

Install the 'Development Tools' Package group, python-devel and sshpass packages:

```
sudo yum group install -y 'Development Tools'
sudo yum -y install python-devel python-virtualenv sshpass
```

Install the OpenStack clients:

```
sudo yum install python-novaclient python-neutronclient python-glanceclient -y
```

Installation

Create or enter the directory where you want to check out the repositories. We assume that both the repositories and your virtual environment are in the same directory. Clone the repositories:

```
git clone https://github.com/redhat-openstack/khaleesi.git
or
git clone https://github.com/redhat-openstack/khaleesi-settings.git
```

read-only mirror:

```
git clone http://<redhat-internal-git-server>/git/khaleesi-settings.git
```

Gerrit:

```
https://review.gerrithub.io/#/q/project:redhat-openstack/khaleesi
```

Create the virtual environment, install ansible, ksgen and kcli utils:

```
virtualenv venv
source venv/bin/activate
pip install ansible==1.9.2
cd khaleesi
cd tools/ksgen
python setup.py install
cd ../kcli
python setup.py install
cd ../..
```

Create the appropriate ansible.cfg for khaleesi:

```
cp ansible.cfg.example ansible.cfg
```

If you are using the OpenStack provisioner, ensure that you have a key uploaded on your remote host or tenant.

Copy the private key file that you will use to access instances to `khaleesi/`. We're going to use the common `example.key.pem` key.:

```
cp ../khaleesi-settings/settings/provisioner/openstack/site/qeos/tenant/keys/example.
↪key.pem  <dir>/khaleesi/
chmod 600 example.key.pem
```

Overview

To use Khaleesi you will need to choose an installer as well as onto which type of provisioner you wish to deploy.The provisioners correspond to the remote machines which will host your environment. Khaleesi currently provide five installers and ten provisioners. For all combinations, the settings are provided by khaleesi-settings through ksgen tool. You will find configuration variables under in the *settings* directory:

settings:

```
|-- provisioner
|    |-- beaker
|    |-- centosci
|    |-- ec2
|    |-- foreman
|    |-- libvirt
|    |-- manual
|    |-- openstack
|    |-- openstack_virtual_baremetal
|    |-- rackspace
|    |-- vagrant
|-- installer
|    |-- devstack
|    |-- opm
|    |-- packstack
|    |-- project
|    |-- rdo_manager
|-- tester
|    |-- api
|    |-- component
|    |-- functional
|    |-- integration
|    |-- pep8
|    |-- rally
|    |-- rhosqe
|    |-- tempest
|    |-- unittest
|-- product
|    |-- rdo
|    |-- rhos
|-- distro
```

One of Khaleesi's primary goals is to break everything into small units. Let's use the installer directory as an example to describe how the configuration tree is built.

Using ksgen with the following flags:

```
--installer=packstack \
--installer-topology=multi-node \
--installer-network=neutron \
--installer-network-variant=ml2-vxlan \
--installer-messaging=rabbitmq \
```

When ksgen reads **–installer=packstack**, it will locate the *packstack.yml* file located within the *settings/installer* directory.

next it goes down the tree to the directory *settings/packstack/topology/multi-node.yml* (because of the flag –installer-topology=multi-node), *settings/packstack/network/neutron.yml*, etc (according to the additional flags) and list all yml files it finds within those directories.

Then ksgen starts merging all YAML files using the parent directories as a base. This means that *packstack.yml* (which holds configuration that is common to packstack) will be used as base and be merged with *settings/packstack/topology/multi-node.yml*, *settings/packstack/network/neutron.yml*, and so on.

Usage

Once everything is set up we can see machines are created using either the rdo-manager or packstack installer. In both cases we're going to use ksgen to supply Khaleesi's ansible playbooks with a correct configuration file.

Installing rdo-manager with the manual provisioner

Here, we will deploy using the RDO-Manager provisioner and manual installer.

First, we create the appropriate configuration file with ksgen. Make sure that you are in your virtual environment that you previously created.

```
source venv/bin/activate
```

Export the ip or fqdn hostname of the test box you will use as the virtual host for osp-director:

```
export TEST_MACHINE=<ip address of baremetal virt host>
```

Generate the configuration with the following command:

```
ksgen --config-dir settings generate \
    --provisioner=manual \
    --product=rdo \
    --product-version=liberty \
    --product-version-build=last_known_good \
    --product-version-repo=delorean_mgt \
    --distro=centos-7.0 \
    --installer=rdo_manager \
    --installer-env=virthost \
    --installer-images=import_rdo \
    --installer-network-isolation=none \
    --installer-network-variant=ml2-vxlan \
    --installer-post_action=none \
    --installer-topology=minimal \
    --installer-tempest=smoke \
    --workarounds=enabled \
    --extra-vars @../khaleesi-settings/hardware_environments/virt/network_configs/
↪none/hw_settings.yml \
    ksgen_settings.yml
```

Note: The “base_dir” key is defined by either where you execute ksgen from or by the $WORKSPACE environment variable. The base_dir value should point to the directory where khaleesi and khaleesi-settings have been cloned.

The result is a YAML file collated from all the small YAML snippets from `khaleesi-settings/settings` (as described in ksgen). All the options are quite self-explanatory and changing them is simple. The rule file is currently only used for deciding the installer+product+topology configuration. Check out ksgen for detailed documentation.

The next step will run your intended deployment:

```
ansible-playbook -vv --extra-vars @ksgen_settings.yml -i local_hosts playbooks/full-
↪job-no-test.yml
```

If any part fails, you can ask for help on the freenode #rdo channel. Don’t forget to save the relevant error lines on something like pastebin.

Using your new undercloud / overcloud

When your run is complete (or even while it’s running), you can log in to your test machine:

```
ssh root@<test_machine>
su stack
```

If you want to log to your new undercloud machine

```
ssh -F ssh.config.ansible undercloud
```

Here you could play with your newly created Overcloud

Installing rdo-manager with centosci provisioner

Here the installation is similiar to *manual* but we will use the centosci provisioner. Notice the changes into the configuration for ksgen:

```
ksgen --config-dir settings generate \
    --provisioner=centosci \
    --provisioner-site=default \
    --provisioner-distro=centos \
    --provisioner-distro-version=7 \
    --provisioner-site-user=rdo \
    --product=rdo \
    --product-version=liberty \
    --product-version-build=last_known_good \
    --product-version-repo=delorean_mgt \
    --distro=centos-7.0 \
    --installer=rdo_manager \
    --installer-env=virthost \
    --installer-images=import_rdo \
    --installer-network-isolation=none \
    --installer-network-variant=ml2-vxlan \
    --installer-post_action=none \
    --installer-topology=minimal \
    --installer-tempest=smoke \
    --workarounds=enabled \
    --extra-vars @../khaleesi-settings/hardware_environments/virt/network_configs/
↪none/hw_settings.yml \
ksgen_settings.yml
```

If any part fails, you can ask for help on the internal #rdo-ci channel. Don't forget to save the relevant error lines on something like pastebin.

Using your new undercloud / overcloud

When your run is complete (or even while it's running), you can log in to your host

```
ssh root@$HOST
su stack
```

If you want to log to your new undercloud machine, just make on your host:

```
ssh -F ssh.config.ansible undercloud
```

Here you could play with your newly created Overcloud

Installing Openstack on Bare Metal via Packstack

All the steps are the same as the All-in-one case. The only difference is running the ksgen with different parameters: Please change the below settings to match your environment:

```
ksgen --config-dir settings generate \
--provisioner=foreman \
--provisioner-topology="all-in-one" \
--distro=rhel-7.1 \
--product=rhos \
--product-version=7.0 \
--product-version-repo=puddle \
--product-version-build=latest \
--extra-vars=provisioner.nodes.controller.hostname=myserver.example.com \
--extra-vars=provisioner.nodes.controller.network.interfaces.external.label=enp4s0f1 \
--extra-vars=provisioner.nodes.controller.network.interfaces.external.config_params.
↪device=enp4s0f1 \
--extra-vars=provisioner.nodes.controller.network.interfaces.data.label="" \
--extra-vars=provisioner.nodes.controller.network.interfaces.data.config_params.
↪device="" \
--extra-vars=provisioner.network.network_list.external.allocation_start=192.168.100.1␣
↪\
--extra-vars=provisioner.network.network_list.external.allocation_end=192.168.100.100␣
↪\
--extra-vars=provisioner.network.network_list.external.subnet_gateway=192.168.100.101␣
↪\
--extra-vars=provisioner.network.network_list.external.subnet_cidr=192.168.100.0/24 \
--extra-vars=provisioner.network.vlan.external.tag=10 \
--extra-vars=provisioner.remote_password=mypassword \
--extra-vars=provisioner.nodes.controller.rebuild=yes \
--extra-vars=provisioner.key_file=/home/user1/.ssh/id_rsa \
--installer=packstack \
--installer-network=neutron \
--installer-network-variant=ml2-vxlan \
--installer-messaging=rabbitmq \
ksgen_settings.yml
```

And then simply run:

```
ansible-playbook -vv --extra-vars @ksgen_settings.yml -i local_hosts playbooks/full-
↪job-no-test.yml
```

Installing Openstack on Bare Metal via rdo-manager

To deploy OpenStack RDO with rdo-manager you will need: - an Undercloud: an existing machine running CentOS 7 since we use rdo-manager, OSP-director requires RHEL7 instead - a set of computer featuring power management interface supported by _Ironic: .. http://docs.openstack.org/developer/tripleo-docs/environments/baremetal.html#ironic-drivers - the undercloud machine must be able to reach the power management interfaces IP - a hardware_environments in khaleesi settings as described below.

Testing Openstack Components

OpenStack components have various set of tests that are referred to as testers. Below is a list of all testers that are supported in khaleesi, for running tests on OpenStack components.The list is sorted by complexity of setting up the environment needed for running the tests:

- pep8
- Unit tests
- Functional
- Integration
- API (in component repo)
- Tempest

Testers are passed to the ksgen CLI as '–tester=' argument value: pep8, unittest, functional, integration, api, tempest

Requirements:

There is only one requirement and it's to have an jenkins-config yml file in the root of the component directory. For example, if the component is neutron, then there should be an neutron/jenkins-config.yml file. The name may differ and can be set by using –extra-vars tester.component.config_file in ksgen invocation.

The structure of an jenkins-config should be similar to:

——————— jenkins-config sample beginning———————— # Khaleesi will read and execute this section only if –tester=pep8 included in ksgen invocation pep8:

> rpm_deps: [python-neutron, python-hacking, pylint] remove_rpm: [] run: tox –sitepackages -v -e pep8 2>&1 | tee ../logs/testrun.log;

# Khaleesi will read and execute this section only if –tester=unittest included in ksgen invocation unittest:

> rpm_deps: [python-neutron, python-cliff] remove_rpm: [] run: tox –sitepackages -v -e py27 2>&1 | tee ../logs/testrun.log;

# Common RPMs that are used by all the testers rpm_deps: [gcc, git, "{{ hostvars[inventory_hostname][tester.component.tox_target]['rpm_deps'] }}"]

# The RPMs that shouldn't be installed when running tests, no matter which tester chosen remove_rpm: ["{{ hostvars[inventory_hostname][tester.component.tox_target]['remove_rpm'] }}"]

# Any additional repos besides defaults that should be enabled to support testing # the repos need to be already installed. this just allows you to enable them. add_additional_repos: [ ]

# Any repos to be disabled to support testing # this just allows you to disable them. remove_additional_repos: [ ]

# Common pre-run steps for all testers neutron_virt_run_config:

> **run: >** set -o pipefail; rpm -qa > installed-rpms.txt; truncate –size 0 requirements.txt && truncate –size 0 test-requirements.txt; {{ hostvars[inventory_hostname][tester.component.tox_target]['run'] }}

# Files to archive

> **archive:**
>
> - ../logs/testrun.log
> - installed-rpms.txt

# Main section that will be read by khaleesi test_config:

> **virt:**

RedHat-7:

setup: enable_repos: “{{ add_additional_repos }}” # Optional. When you would like to look in additional places for RPMs disable_repos: “{{ remove_additional_repos }}” # Optional. When you would like to remove repos to search install: “{{ rpm_deps }}” # Optional. When you would like to install requirements remove: “{{ remove_rpm }}” # Optional. When you would like to remove packages

run: “{{ neutron_virt_run_config.run }}” # A must. The actual command used to run the tests archive: “{{ neutron_virt_run_config.archive }}” # A must. Files to archive

———————— jenkins-config sample end ————————

Usage:

Below are examples on how to use the different testers:

To run pep8 you would use the following ksgen invocation:

ksgen –config-dir settings generate

–provisioner=openstack –provisioner-site=qeos –product=rhos –distro=rhel-7.2 –installer=project –installer-component=nova # OpenStack component on which tests will run –tester=pep8

ksgen_settings.yml

To run unit tests you would use the following ksgen invocation:

ksgen –config-dir settings generate

–provisioner=openstack –provisioner-site=qeos –product=rhos –distro=rhel-7.2 –installer=project –installer-component=cinder –tester=unittest

ksgen_settings.yml

To run functional tests, you would use:

ksgen –config-dir settings generate

–provisioner=openstack –provisioner-site=qeos –distro=rhel-7.2 –product=rhos –installer=project –installer-component=heat –tester=functional

ksgen_settings.yml

To run API in-tree tests, you would use:

ksgen –config-dir settings generate

–provisioner=openstack –provisioner-site=qeos –distro=rhel-7.2 –product=rhos –installer=packstack –installer-config=basic_glance –tester=api –installer-component=glance

ksgen_settings.yml

To run tempest tests, use this invocation:

ksgen –config-dir settings generate

–provisioner=openstack –provisioner-site=qeos –distro=rhel-7.2 –product=rhos –installer=packstack –installer-config=basic_cinder –tester=tempest –tester-tests=cinder_full # For single component tests use cinder_full –tester-setup=git # To install with existing package, use ‘rpm’

ksgen_settings.yml

Key differences between testers:

- pep8 and unittest Do not require installed OpenStack environment while other testers does.

  That is why for the pep8 and unittest the installer is the actual project repo: '–installer=project'.

  For pep8 and unittest khaleesi run would look like this: provision -> install component git repo -> run tests.

  For any other tester khaleesi run would look like this: provision -> install OpenStack* -> copy tests to tester node -> run tests.

  - Using packstack or rdo-manager

- Tempest Holds all the tests in separate project.

  While any other testser can only run on single component, Tempest holds system wide tests that can test multiple component in single run. That's why you need to define the tests to run with '–tester-tests=neutron_full' in order to run single component tests. To run all tests simply use '–tester-tests=all'.

  Tempest itself, as tester framework, can be installed via source or rpm. You can control it with '–tester-setup=git' or '–tester-setup=rpm'.

For all testers khaleesi is collecting logs. There are two type of logs:

1. The actual tests run. Those are subunit streams that khaleesi converts to junitxml.
2. Any additional files that are defined by user for archiving.

Note that pep8 doesn't generate subunit stream, so in this case, the tests logs are simply capture of output redirection from running the tests and not the subunit stream itself.

CHAPTER 2

The hardware_environments

This directory will describe your platform configuration. It comes with the following files:

- network_configs/bond_with_vlans/bond_with_vlans.yml: The network configuration, here *bond_with_vlans* is the name of our configuration, adjust the name for your configuration. You can also prepare a different network profile.
- hw_settings.yml: the configuration to pass to rdo-manager (floating_ip range, neutron internal vlan name, etc)
- vendor_specific_setup: this file is a shell script that will be use to pass extra configuration to your hardware environment (RAID or NIC extract configuration). The file must exist but can be just empty.
- instackenv.json: The list of the power management interfaces. The file is documented in rdo-manager documentation: .. https://repos.fedorapeople.org/repos/openstack-m/rdo-manager-docs/liberty/environments/baremetal.html#instackenv-json

You can find some configuration samples in the khaleesi-settings project: .. https://github.com/redhat-openstack/khaleesi-settings/tree/master/hardware_environments

CHAPTER 3

Start your deployment

This is an example of a ksgen command line, adjust it to match your environment:

```
ksgen --config-dir=settings generate
--provisioner=manual \
--installer=rdo_manager \
--installer-deploy=templates \
--installer-env=baremetal \
--installer-images=import_rdo \
--installer-network=neutron \
--installer-network-isolation=bond_with_vlans \
--installer-network-variant=ml2-vxlan \
--installer-post_action=default \
--installer-topology=minimal \
--installer-tempest=minimal \
--installer-updates=none \
--distro=centos-7.0 \
--product=rdo \
--product-version-build=last_known_good \
--product-version-repo=delorean_mgt \
--product-version=liberty \
--workarounds=enabled \
--extra-vars @/khaleesi_project/khaleesi-settings/hardware_environments/my_test_lab/
↪hw_settings.yml \
/khaleesi_project/ksgen_settings.yml
```

Declare the *$TEST_MACHINE* environment. It should point on the IP of our Undercloud. You should also be able to open a SSH connection as root:

```
export TEST_MACHINE=<ip address of baremetal undercloud host>
ssh root@$TEST_MACHINE
# exit
```

You must create a new *local_host* file. Here again adjust the IP address of your Undercloud:

```
cat <<EOF > local_hosts
[undercloud]
undercloud groups=undercloud ansible_ssh_host=<ip address of baremetal undercloud␣
↪host> ansible_ssh_user=stack ansible_ssh_private_key_file=~/.ssh/id_rsa
[local]
localhost ansible_connection=local
EOF
```

You can now call Khaleesi:

```
ansible-playbook -vv --extra-vars @ksgen_settings.yml -i local_hosts playbooks/full-
↪job-no-test.yml
```

Cleanup

After you finished your work, you can simply remove the created instances by:

```
ansible-playbook -vv --extra-vars @ksgen_settings.yml -i hosts playbooks/cleanup.yml
```

CHAPTER 4

Community Guidelines:

Blueprints:

What is a blueprint?

1. Any new feature requires a blueprint[1].
2. A new feature is anything that changes API / structure of current code and requires a change that spans for more than one file.

   [1] - https://wiki.openstack.org/wiki/Blueprints#Blueprints_reference

Where should one submit blueprints?

1. Any new blueprint requires a discussion in the ML / weekly sync. (we encourage everyone who is involved with the project to join)
2. A code sample / review of a blueprint should use the *git-review -D* for publishing a draft on gerrit.
3. Since the review process is being done as a draft, it is possible to submit the draft prior to an actual ML e-mail.

Reviews:

1. https://review.gerrithub.io/Documentation/config-labels.html#label_Code-Review
2. A "-1"/"-2" from a core requires special attention and a patch should not be merged prior to having the same core remove the "-1"/"-2".
3. In case of a disagreement between two cores, the matter will be brought into discussion on the weekly sync / ML where each core will present his / her thoughts.
4. Self reviews are not allowed. You are required to have at least one more person +1 your code.
5. No review should be merged prior to all gates pass.

6. Bit Rot. To keep the review queue clean an auto-abandoning of dead or old reviews is implemented. A dead review is defined by there are no comments, votes, activity for some agreed upon length of time. Warning will be posted on the review for two weeks of no activity, after the third week the review will be abandoned.

Gates:

1. If a gate has failed, we should first fix that gate and rerun the job to get it passing.
2. When a gate fails due to an infrastructure problem (example: server timeout, failed cleanup, etc), two cores approval is required in order to remove a gate "-1" vote

Commits:

1. Each commit should be dedicated to a specific subject and not include several patches that are not related.
2. Each commit should have a detailed commit message that describes the "high level" of what this commit does and have reference to other commits in case there is a relationship.

Cores:

1. Need to have quality reviews.
2. Reviews are well formed, descriptive and constructive.
3. Reviews are well thought and do not result in a followed revert. (often)
4. Should be involved in the project on a daily basis.

CHAPTER 5

Khaleesi Best Practices Guide

The purpose of this guide is to lay out the coding standards and best practices to be applied when working with Khaleesi. These best practices are specific to Khlaeesi but should be in line with general Ansible guidelines.

Each section includes:

- A 'Rule' which states the best practice to apply
- Explanations and notable exceptions
- Examples of code applying the rule and, if applicable, examples of where the exceptions would hold

General Best Practices

Rule: Whitespace and indentation - Use 4 spaces.

Ensure that you use 4 spaces, not tabs, to separate each level of indentation.

Examples:

```
# BEST_PRACTICES_APPLIED
- name: set plan values for plan based ceph deployments
  shell: >
      source {{ instack_user_home }}/stackrc;
      openstack management plan set {{ overcloud_uuid }}
          -P Controller-1::CinderEnableIscsiBackend=false;
  when: installer.deploy.type == 'plan'
```

Rule: Parameter Format - Use the YAML dictionary format when 3 or more parameters are being passed.

When several parameters are being passed in a module, it is hard to see exactly what value each parameter is getting. It is preferable to use the Ansible YAML syntax to pass in parameters so that it is clear what values are being passed for each parameter.

Examples:

```
# Step with all arguments passed in one line
- name: create .ssh dir
  file: path=/home/{{ provisioner.remote_user }}/.ssh mode=0700 owner=stack␣
↪group=stack state=directory

# BEST_PRACTICE_APPLIED
- name: create .ssh dir
  file:
      path: /home/{{ provisioner.remote_user }}/.ssh
      mode: 0700
      owner: stack
      group: stack
      state: directory
```

Rule: Line Length - Keep text under 100 characters per line.

For ease of readability, keep text to a uniform length of 100 characters or less. Some modules are known to have issues with multi-line formatting and should be commented on if it is an issue within your change.

Examples:

```
# BEST_PRACTICE_APPLIED
- name: set plan values for plan based ceph deployments
  shell: >
      source {{ instack_user_home }}/stackrc;
      source {{ instack_user_home }}/deploy-nodesrc;
      openstack management plan set {{ overcloud_uuid }}
          -P Controller-1::CinderEnableIscsiBackend=false
          -P Controller-1::CinderEnableRbdBackend=true
          -P Controller-1::GlanceBackend=rbd
          -P Compute-1::NovaEnableRbdBackend=true;
  when: installer.deploy.type == 'plan'

# EXCEPTION: - When a module breaks from multi-line use, add a comment to indicate it
# The long line in this task fails when broken down
- name: copy over common environment file (virt)
  local_action: >
      shell pushd {{ base_dir }}/khaleesi; rsync --delay-updates -F --compress --
↪archive --rsh \
      "ssh -F ssh.config.ansible -S none -o StrictHostKeyChecking=no" \
      {{base_dir}}/khaleesi-settings/hardware_environments/common/plan-parameter-
↪neutron-bridge.yaml undercloud:{{ instack_user_home }}/plan-parameter-neutron-
↪bridge.yaml
```

Rule: Using Quotes - Use single quotes.

Use single quotes throughout playbooks except when double quotes are required for `shell` commands or enclosing `when` statements.

Examples:

```
# BEST_PRACTICE_APPLIED
- name: get floating ip address
  register: floating_ip_result
  shell: >
      source {{ instack_user_home }}/overcloudrc;
      neutron floatingip-show '{{ floating_ip.stdout }}' | grep 'ip_address' | sed -e
↪'s/|//g';

# EXCEPTION - shell command uses both single and double quotes
- name: copy instackenv.json to root dir
  shell: >
      'ssh -t -o "StrictHostKeyChecking=no" {{ provisioner.host_cloud_user }}@{{␣
↪floating_ip.stdout }} \
      "sudo cp /home/{{ provisioner.host_cloud_user }}/instackenv.json /root/
↪instackenv.json"'
  when: provisioner.host_cloud_user != 'root'

# EXCEPTION - enclosing a ``when`` statement
- name: copy instackenv.json to root dir
  shell: >
      'ssh -t -o "StrictHostKeyChecking=no" {{ provisioner.host_cloud_user }}@{{␣
↪floating_ip.stdout }} \
      "sudo cp /home/{{ provisioner.host_cloud_user }}/instackenv.json /root/
↪instackenv.json"'
  when: "provisioner.host_cloud_user != {{ user }}"
```

Rule: Order of Arguments - Keep argument order consistent within a playbook.

The order of arguments is:

```
tasks:
  - name:
    hosts:
    sudo:
    module:
    register:
    retries:
    delay:
    until:
    ignore_errors:
    with_items:
    when:
```

> **Warning:** While `name` is not required, it is an Ansible best practice, and a Khaleesi best practice, to name all tasks.

Examples:

```
# BEST_PRACTICE_APPLIED - polling
- name: poll for heat stack-list to go to COMPLETE
  shell: >
      source {{ instack_user_home }}/stackrc;
      heat stack-list;
  register: heat_stack_list_result
  retries: 10
  delay: 180
  until: heat_stack_list_result.stdout.find("COMPLETE") != -1
  when: node_to_scale is defined

# BEST_PRACTICE_APPLIED - looping through items
- name: remove any yum repos not owned by rpm
  sudo: yes
  shell: rm -Rf /etc/yum.repos.d/{{ item }}
  ignore_errors: true
  with_items:
      - beaker-*
```

Rule: Adding Workarounds - Create bug reports and flags for all workarounds.

More detailed information and examples on working with workarounds in Khaleesi can be found in the documentation on Handling Workarounds.

Rule: Ansible Modules - Use Ansible modules over `shell` where available.

The generic `shell` module should be used only when there is not a suitable Ansible module available to do the required steps. Use the `command` module when a step requires a single bash command.

Examples:

```
# BEST_PRACTICE_APPLIED - using Ansible 'git' module rather than 'shell:  git clone'
- name: clone openstack-virtual-baremetal repo
  git:
      repo=https://github.com/cybertron/openstack-virtual-baremetal/
      dest={{instack_user_home}}/openstack-virtual-baremetal

# BEST_PRACTICE_APPLIED - using Openstack modules that have checks for redundancy or
# existing elements
- name: setup neutron network for floating ips
  register: public_network_uuid_result
  quantum_network:
      state: present
      auth_url: '{{ get_auth_url_result.stdout }}'
      login_username: admin
      login_password: '{{ get_admin_password_result.stdout }}'
      login_tenant_name: admin
      name: '{{ installer.network.name }}'
      provider_network_type: '{{ hw_env.network_type }}'
      provider_physical_network: '{{ hw_env.physical_network }}'
      provider_segmentation_id: '{{ hw_env.ExternalNetworkVlanID }}'
      router_external: yes
      shared: no

# EXCEPTION - using  shell as there are no Ansible modules yet for updating nova␣
↪quotas
- name: set neutron subnet quota to unlimited
  ignore_errors: true
  shell: >
      source {{ instack_user_home }}/overcloudrc;
```

```
   neutron quota-update --subnet -1;
   neutron quota-update --network -1;
```

Rule: Scripts - Use scripts rather than shell for lengthy or complex bash operations.

Scripts can hide output details and debugging scripts requires the user to look in multiple directories for the code involved. Consider using scripts over `shell` if the step in Ansible requires multiple lines (more than ten), involves complex logic, or is called more than once.

Examples:

```
# BEST_PRACTICE_APPLIED - calling Beaker checkout script,
# keeps the complexity of Beaker provisioning in a standalone script
- name: provision beaker machine with kerberos auth
  register: beaker_job_status
  shell: >
     chdir={{base_dir}}/khaleesi-settings
     {{base_dir}}/khaleesi-settings/beakerCheckOut.sh
     --arch={{ provisioner.beaker_arch }}
     --family={{ provisioner.beaker_family }}
     --distro={{ provisioner.beaker_distro }}
     --variant={{ provisioner.beaker_variant }}
     --hostrequire=hostlabcontroller={{ provisioner.host_lab_controller }}
     --task=/CoreOS/rhsm/Install/automatjon-keys
     --keyvalue=HVM=1
     --ks_meta=ksdevice=link
     --whiteboard={{ provisioner.whiteboard_message }}
     --job-group={{ provisioner.beaker_group }}
     --machine={{ lookup('env', 'BEAKER_MACHINE') }}
     --timeout=720;
  async: 7200
  poll: 180
  when: provisioner.beaker_password is not defined
```

Rule - Roles - Use roles for generic tasks which are applied across installers, provisioners, or testers.

Roles should be used to avoid code duplication. When using roles, take care to use debug steps and print appropriate code output to allow users to trace the source of errors since the exact steps are not visible directly in the playbook run. Please review the Ansibles official best practices documentation for more information regarding role structure.

Examples:

```
# BEST_PRACTICE_APPLIED - validate role that can be used with multiple installers
https://github.com/redhat-openstack/khaleesi/tree/master/roles/validate_openstack
```

RDO-Manager Specific Best Practices

The following rules apply to RDO-Manager specific playbooks and roles.

Rule: Step Placement - Place a step under the playbook directory named for where it will be executed.

The RDO-Manager related playbooks have the following directory structure:

```
|-- installer
     |-- rdo-manager
          |-- overcloud
          |-- undercloud
| -- post-deploy
    |-- rdo-manager
```

These guidelines are used when deciding where to place new steps:

- `undercloud` - any step that can be executed without the overcloud
- `overcloud` - any step that is used to deploy the overcloud
- `post-deploy` - always a standalone playbook - steps executed once the overcloud is deployed

Rule: Idempotency - Any step executed post setup should be idempotent.

RDO-Manager has some set up steps that cannot be run multiple times without cleaning up the environment. Any step added after setup should be able to rerun without causing damage. *Defensive programming* conditions, that check for existence or availability etc. and modify when or how a step is run, can be added to ensure playbooks remain idempotent.

Examples:

```
# BEST_PRACTICE_APPLIED - using Ansible modules that check for existing elements
- name: create provisioning network
  register: provision_network_uuid_result
  quantum_network:
      state: present
      auth_url: "{{ get_auth_url_result.stdout }}"
      login_username: admin
      login_password: "{{ get_admin_password_result.stdout }}"
      login_tenant_name: admin
      name: "{{ tmp.node_prefix }}provision"

# BEST_PRACTICE_APPLIED - defensive programming,
# ignoring errors from creating a flavor that already exists
- name: create baremetal flavor
  shell: >
      source {{ instack_user_home }}/overcloudrc;
      nova flavor-create baremetal auto 6144 50 2;
  ignore_errors: true
```

Applying these Best Practices and Guidelines

Before submitting a review for Khaleesi please review your changes to ensure they follow with the best practices outlined above.

Contributing to this Guide

Additional best practices and suggestions for improvements to the coding standards are welcome. To contribute to this guide, please review contribution documentation and submit a review to GerritHub.

CHAPTER 6

Contributing to Khaleesi development

Getting Started with Khaleesi.

see *Prerequisites*

Associated Settings Repository

https://github.com/redhat-openstack/khaleesi-settings

Help, I can't run this thing

Look under the khaleesi/tools/wrappers directory

Code Review (IMPORTANT)

Pull requests will not be looked at on khaleesi github. Code submissions should be done via gerrithub (https://review.gerrithub.io). Please sign up with https://www.gerrithub.io and your github credentials to make submissions. Additional permissions on the project will need to be done on a per-user basis.

When you set up your account on gerrithub.io, it is not necessary to import your existing khaleesi fork.:

> yum install git-review

To set up your repo for gerrit:

Add a new remote to your working tree:

```
git remote add gerrit ssh://username@review.gerrithub.io:29418/redhat-openstack/
↪khaleesi
```

Replace username with your gerrithub username.

Now run:

```
git review -s
scp -p -P 29418 username@review.gerrithub.io:hooks/commit-msg `git rev-parse --git-
↪dir`/hooks/commit-msg
```

Again, replace username with your gerrithub username.

Required Ansible version

Ansible 1.8.2 is now required.

Std{out,err} callback plugin

To use the callback plugin that will log all stdout, stderr, and other data about most tasks, you must set the ANSIBLE_CALLBACK_PLUGINS envvar. You can also set the KHALEESI_LOG_PATH envvar. KHALEESI_LOG_PATH defaults to /tmp/stdstream_logs.:

> export ANSIBLE_CALLBACK_PLUGINS=$WORKSPACE/khaleesi/plugins/callbacks export KHALEESI_LOG_PATH=$WORKSPACE/ansible_log

Khaleesi use cases

Check khaleesi *Usage*

Handling workarounds

As OpenStack code and also general tools/libraries from distribution may contain bugs responsible for blocking installation, running or behaviour of OpenStack, it can be necessary or very helpfull to implement workaround for the time while the bug is being properly fixed.

Basically any bug has to be **reported first** in corresponding place (eg. Red Hat Bugzilla, Launchpad, ...), idealy with possible workaround described there before implementation in khaleesi happens.

To reflect this there is common style *how to implement workarounds*, to support tracking them and obtaining their *bug status overview*, and examples for *overriding workaround usage* per single run.

How to implement workarounds

1. workaround has to have it's **flag** set **in khaleesi-settings** (per product/version/...):
    - in 'workarounds:{}' top level dict
    - flag value should be also dict, with boolean property **enabled** (true|false)
    - has to be named in format `[bug-tracker-prefix][bug-number-ident]` where prefixes are:
        - *rhbz* for Red Hat Bugzilla

- *lp* for Launchpad
- for example `workarounds.rhbz1138740.enabled=true`

2. it's **implementation** may be:
   - single task (guarded by `when: workarounds | bug(flag)` condition)
   - whole play(s) scoped to group after *group_by* with when condition
   - group name has to be in form *workaround_[flag]* where the flag means the key used in *workarounds* settings dictionary (eg. rhbz1138740). to not cause confusion with regular groups (imagine usage of group names like nova_something etc)

Example of enabling the workaround in settings (same for both following examples):

```
# khaleesi-settings:settings/product/rdo/kilo.yml
workarounds:
    rhbz1138740:
        desc: "Workaround BZ#1138740: Install nova-objectstore for S3 tests"
        enabled: True
    lp0000001:
        enabled: True
```

Example of single-task workaround:

```
# khaleesi:playbooks/.../some.yml
# can be some already existing play

- hosts: somenodes
  tasks:

    # ... snip ... (part of bigger play)
    # you just add one task in the list like following one:

    - fileinline: tempest.conf enable_s3_tests=true
      when: workarounds | bug('rhbz1138740')
```

Example of multi-task workaround Play (utilizing group_by):

```
# khaleesi:playbooks/.../some.yml
# ... snip ... following is what you are adding:

- hosts: controller
  tasks:
    - group_by: key=workaround_rhbz1138740
      when: workarounds| bug('rhbz1138740')

- name: Workaround BZ#1138740 Install Nova Object Store for S3 tests
  hosts: workaround_rhbz1138740
  tasks:
    - yum: ...
    - service: ...
    ...
```

Bug status overview

For getting overview of currently implemented workarounds, which follow this guide, You can use helper tool which detects them in the code and provides more detailed description (fetched from bugtrackers), to see **all workarounds**

present in the settings yamls:

```
$ cd khaleesi
$ ./tools/workaround_status ./settings

... list of bugs, their title, state and url, also in which yamls we enable them ...
```

But as some workarounds can be for older version of OpenStack, and so marked as resolved in newer versions, this will show warning that some of the bugs are already closed (as there are settings for older versions too).

So instead of settings folder, You can point it at the **ksgen_settings.yaml** file to get overview of **workarounds used in specific job run**, where there normally shouldn't be any workaround for closed bug, unless it's just freshly resolved and author of the workaround didn't revalidated it yet:

```
$ ksgen ...
$ ./tools/workaround_status ksgen_settings.yaml

... partial list of information about bugs,
... for just those in use (enabled) for given configuration
```

Overriding workaround usage

Workarounds are enabled by default (based on values in settings), to disable all of them by force use:

```
ksgen ... --extra-vars 'workarounds={}'
```

to disable (or force enable) specific one:

```
ksgen ... --extra-vars '{"workarounds": {"rhbz1138740": {"enabled": true}}}'
```

Variables meaning

- product.build: This variable can come with the following values: *latest*, *ga*, *last_known_good*. It is used to construct the path to the images, like: *product.images.url.rhos.8-directory.latest.7.2*
- product.full_version: The component to test, can be either: - OSP: *7*, *7-director*, *8*, *8-director* or - RDO: *Juno*, *Kilo*, *Liberty*
- product.core_product_version: The major version number of the product, e.g: *7* or *8*.
- product.repo_type: Can be either *puddle* or *poodle* for OSP or *delorean* and *deorean_mgt* for RDO. - puddle: automatic snapshot of the development repositories (core and OSP-d) - poodle: a poodle is a stabilized version of the repository (core only) - delorean: a delorean build result or RDO - delorean_mgt: a delorean build result of RDO-manager (deprecated)
- product.repo.poodle_pin_version: Define a specific version of the poodle. The value can either be *latest* or specify a given version like *2015-12-03.1*. The variable is in use only if *product.repo_type* is *poodle*.
- product.repo.puddle_pin_version: Like for *product.repo.poodle_pin_version* but for a core puddle. The variable is in use only if *product.repo_type* is *poodle*.
- product.repo.puddle_director_pin_version: The OSP-d puddle version to test. Default is *latest*. This variable is enable only if *product.repo_type* is *puddle*.
- product.repo.delorean_pin_version: Pin on this dolorean build hash.

CHAPTER 7

ksgen - Khaleesi Settings Generator

Setup

It's advised to use ksgen in a virtual Python environment.

```
$ virtualenv ansible # you skip this and use an existing one
$ source ansible/bin/activate
$ python setup.py install # do this in the ksgen directory
```

Running ksgen

Assumes that ksgen is installed, else follow *Setup*.

You can get general usage information with the `--help` option. After you built a proper settings directory ("configuration tree") structure, you need to let ksgen know where it is. Invoke ksgen like this to show you all your possible options:

```
ksgen --config-dir <dir> help
```

If `--config-dir` is not provided, ksgen will look for the `KHALEESI_DIR` environment variable, so it is a good practice to define this in your `.bashrc` file:

```
export KHALEESI_DIR=<dir>
ksgen help
```

This displays options you can pass to ksgen to *generate* the all-in-one settings file.

Using ksgen

How ksgen works

ksgen is a simple utility to **merge** dictionaries (hashes, mappings), and lists (sequences, arrays). Any scalar value (string, int, floats) are overwritten while merging.

For e.g.: merging `first_file.yml` and `second_file.yml`

first_file.yml:

```
foo:
  bar: baz
  merge_scalar: a string from first dict
  merge_list: [1, 3, 5]
  nested:
    bar: baz
    merge_scalar: a string from first dict
    merge_list: [1, 3, 5]
```

and second_file:

```
foo:
  too: moo
  merge_scalar: a string from second dict
  merge_list: [6, 2, 4, 3]
  nested:
    bar: baz
    merge_scalar: a string from second dict
    merge_list: [6, 2, 4, 3]
```

produces the output below:

```
foo:
  bar: baz
  too: moo
  merge_scalar: a string from second dict
  merge_list: [1, 3, 5, 6, 2, 4, 3]
  nested:
    bar: baz
    merge_scalar: a string from second dict
    merge_list: [1, 3, 5, 6, 2, 4, 3]
```

Organizing settings files

ksgen requires a `--config-dir` option which points to the directory where the settings files are stored. ksgen traverses the *config-dir* to generate a list of options that sub-commands (`help`, `generate`) can accept.

First level directories inside the config-dir are used as options, and yml files inside them are used as option values. You can add suboptions if you add a directory with the same name as the value (without the extension). Inside that directory, the pattern repeats: you can specify options by creating directories, and inside them yml files.

If the directory name and a yml file don't match, it doesn't add a suboption (it gets ignored). You can use this to store YAMLs for includes.

Look at the following schematic example:

```
settings/
- option1/
|   - ignored/
|   |   - useful.yml
|   - value1.yml
|   - value2.yml
- option2/
    - value3/
    |   - suboption1/
    |       - value5.yml
    |       - value6.yml
    - value3.yml
    - value4.yml
```

The valid settings will be:

```
$ ksgen --config-dir settings/ help
[snip]
 Valid configs are:
    --option1=<val>             [value2, value1]
    --option2=<val>             [value4, value3]
    --option2-suboption1=<val> [value6, value5]
```

A more organic settings example:

```
settings/
- installer/
|   - foreman/
|   |   - network/
|   |       - neutron.yml
|   |       - nova.yml
|   - foreman.yml
|   - packstack/
|   |   - network/
|   |       - neutron.yml
|   |       - nova.yml
|   - packstack.yml
- provisioner/
    - trystack/
    |   - tenant/
    |   |   - common/
    |   |   |   - images.yml
    |   |   - john-doe.yml
    |   |   - john.yml
    |   |   - smith.yml
    |   - user/
    |       - john.yml
    |       - smith.yml
    - trystack.yml
```

ksgen maps all directories to options and files in those directories to values that the option can accept. Given the above directory structure, the options that `generate` can accept are as follows

Options	Values
provisioner	trystack
provisioner-tenant	smith, john, john-doe
provisioner-user	john, smith
installer	packstack, foreman
installer-network	nova, neutron

Note: ksgen skips provisioner/trystack/tenant/common directory since there is no `common.yml` file under the `tenant` directory.

Default settings

Default settings allow the user to supply only the minimal required flags in order to generate a valid output file. Defaults settings will be loaded from the given 'top-level' parameters settings files if they are defined in them. Defaults settings for any 'non top level' parameters that have been given will not been loaded.

Example of defaults section in settings files:

Listing 7.1: provisioner/openstack.yml

```
defaults:
  site: openstack-site
  topology: all-in-one
```

Listing 7.2: provisioner/openstack/site/openstack-site.yml

```
defaults:
  user: openstack-user
```

Usage example:

```
ksgen --config-dir=/settings/dir/path generate --provisioner=openstack settings.yml
```

generate: merges settings into a single file

The `generate` command merges multiple settings file into a single file. This file can then be passed to an ansible playbook. ksgen also allows merging, extending, overwriting (!overwrite_) and looking up (!lookup_) settings that ansible (at present) doesn't allow.

Merge order

Refering back to the *settings example* above, if you execute the command:

```
ksgen --config-dir sample generate \
  --provisioner trystack \
  --installer packstack \
  --provisioner-user john \
  --extra-vars foo.bar=baz \
  --provisioner-tenant smith \
  output-file.yml
```

generate command will create an `output-file.yml` that include all contents of

SL	File	Reason
1	provisioner/trystack.yml	The first command line option
2	merge provisioner/trystack/user/john.yml	The first child of the first command line option
3	merge provisioner/trystack/tenant/smith.yml	The next child of the first command line option
4	merge installer/packstack.yml	the next top-level option
5	add/merge foo.bar: baz. to output	extra-vars get processed at the end

Rules file

ksgen arguments can get quite long and tedious to maintain, the options passed to ksgen can be stored in a rules yaml file to simplify invocation. The command above can be simplified by storing the options in a yaml file.

rules_file.yml:

```
args:
  provisioner: trystack
  provisioner-user: john
  provisioner-tenant: smith
  installer: packstack
  extra-vars:
    - foo.bar=baz
```

ksgen generate using rules_file.yml:

```
ksgen --config-dir sample generate \
  --rules-file rules_file.yml \
  output-file.yml
```

Apart from the **args** key in the rules-files to supply default args to generate, validations can also be added by adding a ‘validation.must_have’ like below:

```
args:
  ...
    default args
  ...
validation:
  must_have:
      - topology
```

The generate commmand would validate that all options in must_have are supplied else it will fail with an appropriate message.

YAML tags

ksgen uses Configure python package to keep the yaml files DRY. It also adds a few yaml tags like !overwrite, !lookup, !join, !env to the collection.

overwrite

Use *overwrite* tag to overwrite value of a key. This is especially useful when to clear the contents of an array and add new one

For e.g.: merging

```
foo: bar
```

and

```
foo: [1, 2, 3]
```

will fail since there is no reasonable way to merge a string and an array. Use overwrite to set the contents of foo to [1, 2, 3] as below

```
foo: !overwrite [1, 2, 3]
```

lookup

Lookup helps keep the yaml files DRY by replacing looking up values for keys.

```
foo: bar
key_foo: !lookup foo
```

After ksgen process the yaml above the value of *key_foo* will be replaced by *bar* resulting in the output below.

```
foo: bar
key_foo: bar
```

This works for several consecutive !lookup as well such as

```
foo:
    barfoo: foobar
bar:
    foo: barfoo

key_foo: !lookup foo[ !lookup bar.foo ]
```

After ksgen process the yaml above the value of *key_foo* will be replaced by *foobar*

Warning: (Limitation) Lookup is done only after all yaml files are loaded and the values are merged so that the entire yaml tree can be searched. This prevents combining other yaml tags with *lookup* as most tags are processed when yaml is loaded and not when it is written. For example:

```
home: /home/john
bashrc: !join [ !lookup home, /bashrc ]
```

This **will fail** to set bashrc to */home/john/bashrc* where as the snippet below will work as expected:

```
bashrc: !join [ !env HOME, /bashrc ]
```

join

Use join tag to join all items in an array into a string. This is quite useful when using yaml anchors or *env* tag.

```
unused:
  baseurl: &baseurl http://foobar.com/repo/

repo:
  epel7: !join[ *baseurl, epel7 ]

bashrc: !join [ !env HOME, /bashrc ]
```

env

Use env tag to lookup value of an environment variable. An optional default value can be passed to the tag. if no default values are passed and the lookup fails, then a runtime KeyError is generated. Second optional argument will reduce length of value by given value

```
user_home: !env HOME
user_shell !env [SHELL, zsh]  # default shell is zsh
job_name_parts:
   - !env [JOB_NAME, 'dev-job']
   - !env [BUILD_NUMBER, None ]
   - !env [USER, None, 5]

job_name: "{{ job_name_parts | reject(none) | join('-') }}"
```

The snippet above effectively uses *env* tag and default option to set the *job_name* variable to *$JOB_NAME-$BUILD_NUMBER-${USER:0:5}* if they are defined else to 'dev-job'.

limit_chars

This function will trim value of variable or string to given length.

> **debug:** message: !limit_chars ['some really looong text' 10]

Debugging errors in settings

ksgen is heavily logged and by default the log-level is set to **warning**. Changing the debug level using the `--log-level` option to **info** or **debug** reveals more information about the inner workings of the tool and how values are loaded from files and merged.

Developing ksgen

Running ksgen unit-tests

```
pip install pytest
py.test tests/test_<filename>.py
# or
python tests/test_<filename>.py  <method_name>
```

CHAPTER 8

kcli - Khaleesi CLI tool

`kcli` is intended to reduce Khaleesi users' dependency on external CLI tools.

Setup

Note: Khaleesi is based on ansible so for setup to work, `kcli` requires ansible installed:

```
$ pip install ansible
```

from khaleesi directory.

```
$ cd tools/kcli
$ python setup.py install # do this in the ``kcli`` directory
```

Running kcli

Assumes that `kcli` is installed, else follow *Setup*.

You can get general usage information with the `--help` option:

```
kcli --help
```

This displays options you can pass to `kcli`.

KCLI execute

Note: This is a wrapper for the `ansible-playbook` command. In verbose mode, the equivalent anisble command will be printed.

Executes pre-configured ansible-playbooks, with given settings YAML file generated by ksgen. if no settings file is defined, will look for the default name `ksgen_settings.yml`:

```
kcli [-vvvv] [--settings SETTINGS] execute [-i INVENTORY] [--provision] [--install] [-
↪-test] [--collect-logs] [--cleanup]
```

CHAPTER 9

Handling the Jenkins job definitions

This section deals with the issue of adding, removing and changing of job definitions through the JJB files.

A general documentation about JJB can be found on its website. When in doubt about what an option means in the job description, search in this manual.

Location and structure

The job definitions reside in `khaleesi-settings/jobs` and they are in YAML format. The changes are applied on the official Jenkins server by submitting a change to the files in this repository, and running the jenkins_job_builder job.

If you removed some job, please make sure to disable or delete the job that is no longer used. The job has a diff output at the end of the run that compares the jobs that exist on the server but are not part of the job definitions.

The `defaults.yaml` file containts the default values that all jobs get by default. It also contains some macros that can be referenced later. You probably don't need to modify this file.

At the moment the `main.yaml` file contains the definitions for all RDO CI jobs. On the top of the file you find a **job-template** that is the base of our jobs. You can see that its name contains a lot of variables in curly brackets "{}", they are replaced by the actual job definitions, and we give them values by the **project** definitions lower in the file.

The project definitions are creating a matrix of variables, from which all the possible combinations get created on the Jenkins server.

Adding new jobs

The need for a new job could arise when we want to extend our testing. There's a significant difference between two cases:

- adding a new value to an existing ksgen option (can be thought of as extending the testing matrix in an existing dimension)

- adding a new option to ksgen (adding a new dimension to the testing matrix)

The first case is significantly easier to deal with, so let’s discuss that first.

Let’s say you added the new variable *foo* for the **distro** setting. If you want to create a whole new set of jobs, then you might want to create a new **project** definition. In most cases, it’s enough if you extend an existing definition. In that case, just add the relevant option to the proper place. Here’s an example:

```
- project:
    name: rhos7-jobs
    product:
            - rhos
        product-version:
            - 7_director
        product-version-repo:
            - poodle
            - puddle
        distro:
            - rhel-7.1
            - foo
        messaging:
            - rabbitmq
[the rest of the definition is omitted]
```

If you want to extend the matrix, the changes are more numerous.

- the **job-template** has to be changed, and the new option added to the name
- the *ksgen-builder* macro needs alteration, both in the calling in the job template, and in the shell script part (add it to the ksgen command).
- add the option to all the project definitions that are using the template (currently all of them), modifying the template name in them too.

This will also result in a replacement of all the Jenkins jobs that use the template, as the naming changes.

CHAPTER 10

Creating a Jenkins server with Khaleesi jobs

Getting a Jenkins

Deploying the jobs require a properly configured Jenkins server. We have a couple of them already, but if you want to experiment without any fear of messing with other jobs, the best is to get yourself a server. It's recommended to use the Long Term Support (LTS) version.

You can create a VM on any of our OpenStack instances (don't forget to use your public key for it), attach a floating IP and then install Jenkins. This should work both on Fedora and RHEL:

```
sudo wget -O /etc/yum.repos.d/jenkins.repo \
http://pkg.jenkins-ci.org/redhat-stable/jenkins.repo
sudo rpm --import \
http://pkg.jenkins-ci.org/redhat-stable/jenkins-ci.org.key
yum install jenkins
service jenkins start
chkconfig jenkins on
```

Installing plugins

Our jobs require quite a few plugins. So when your Jenkins is up and running, navigate to `http://$JENKINS_IP:8080/cli` and download jenkins-cli.jar.

Afterwards. just execute these commands:

```
java -jar jenkins-cli.jar -s http://$JENKINS_IP:8080/ install-plugin git \
xunit ansicolor multiple-scms rebuild ws-cleanup gerrit-trigger \
parameterized-trigger envinject email-ext sonar copyartifact timestamper \
build-timeout jobConfigHistory test-stability jenkins-multijob-plugin \
dynamicparameter swarm shiningpanda scm-api ownership mask-passwords \
jobConfigHistory buildresult-trigger test-stability dynamicparameter \
scm-api token-macro swarm scripttrigger groovy-postbuild shiningpanda \
jenkins-multijob-plugin ownership
```

Deploying the jobs

You can do this from any machine. Install JJB:

```
pip install jenkins-job-builder
```

Create a config file for it:

```
cat > my_jenkins << EOF
[jenkins]
user=my_username
password=my_password
url=http://$JENKINS_IP:8080/
EOF
```

Optional: I recommend turning off the timed runs (deleting - timed lines from the job template), otherwise they would run periodically on your test server:

```
sed '/- timed:/d' khaleesi-settings/jobs/main.yaml
```

Then just run the job creation (the last argument is the job directory of the khaleesi-settings repo, which I assume you cloned previously):

```
jenkins-jobs --conf my_jenkins update khaleesi-settings/jobs/
```

Bonus: Test your job changes

If you want to experiment with your own job changes:

- open `khaleesi-settings/jobs/defaults.yaml`
- **change the khaleesi and/or khalessi-settings repo URL to your own** and your own branch
- execute the job building step above

Now your test server will use your own version of the repos.

Tip: you can `git stash save testing` these changes, and later recall them with `git stash pop` to make this testing step easy along the code review submission.

Creating a Jenkins slave

Now you need to either set up the machine itself as a slave, or attach/create a slave to run the jobs. The slave needs to have the 'khaleesi' label on it to run the JJB jobs.

You can set up a slave with the help of the khaleesi-slave repo.

```
git clone git@github.com:redhat-openstack/khaleesi-settings.git
cd khaleesi-settings/jenkins/slaves
cat << EOF > hosts
$SLAVE_IP

[slave]
$SLAVE_IP
EOF
```

Check the settings in ansible.cfg.sample. If you run into weird ansible errors about modules you probably don't have them set up correctly. This should be enough:

```
[defaults]
host_key_checking = False
roles_path = ./roles
```

Execute the playbook, assuming that your instance uses the "fedora" user and you can access it by the "rhos-jenkins.pem" private key file. If you used a proper cloud image, it will fail.

```
ansible-playbook -i hosts -u fedora playbooks/basic_internal_slave.yml --private-
↪key=rhos-jenkins.pem -v
```

Login to the machine, become root and delete the characters from /root/.ssh/authorized_keys before the "ssh-rsa" word. Log out and rerun the ansible command. It should now finish successfully.

Add the slave to Jenkins. If you used the same machine, specify `localhost` and add the relevant public key for the rhos-ci user. use the `/home/rhos-ci/jenkins` directory, add the `khaleesi` label, only run tied jobs. You're done.

Jenkins RDO-Manager:

For using khaleesi with Jenkins, first of all see the steps *Getting a Jenkins* part for setting up a Jenkins slave and for use jjb.

If you want to setup a manual job on Jenkins you have to follow those steps:

Setup a slave (General):

Check the option:

```
Restrict where this project can be run
```

And put the name of your slave.

Clone the repositories (Source Code Management):

Select the choice:

```
Multiple SCMs
```

And put the urls of the khaleesi / khaleesi-settings repositories. You need to specify to jenkins to checkout the repositories in a sub-directory:

```
Check out to a sub-directory
```

And specify for each:

```
khaleesi
khaleesi-settings
```

Build Environment:

Check the option:

> Delete workspace before build starts

Build:

Add a step:

```
Virtualenv Builder
```

And select:

```
Python version: System-CPython-2.7
Nature: Shell
```

And put the above informations into the shell command:

```
pip install -U ansible==1.9.2 > ansible_build; ansible --version
source khaleesi-settings/jenkins/ansible_rdo_mang_settings.sh

# install ksgen
pushd khaleesi/tools/ksgen
python setup.py install
popd

pushd khaleesi
# generate config
ksgen --config-dir=../khaleesi-settings/settings generate \
    --provisioner=your_provisioner (see cookbook)

# get nodes and run test
set +e
anscmd="stdbuf -oL -eL ansible-playbook -vv --extra-vars @ksgen_settings.yml"

$anscmd -i local_hosts playbooks/full-job-no-test.yml
result=$?

infra_result=0
$anscmd -i hosts playbooks/collect_logs.yml &> collect_logs.txt || infra_result=1
$anscmd -i local_hosts playbooks/cleanup.yml &> cleanup.txt || infra_result=2

if [[ "$infra_result" != "0" && "$result" = "0" ]]; then
    # if the job/test was ok, but collect_logs/cleanup failed,
    # print out why the job is going to be marked as failed
    result=$infra_result
    cat collect_logs.txt
    cat cleanup.txt
fi

exit $result
```

Post-build actions:

Add a post build action for collecting logs and required files for debuging and archived them:

```
Archive the artifacts: **/collected_files/*.tar.gz, **/nosetests.xml, **/ksgen_
↪settings.yml
```

If you run tempest during the deployment add the following step for collecting the tests result:

```
Publish JUnit test result report
Test Report XMLs : **/nosetests.xml
Check : Test stability history
```

CHAPTER 11

Khaleesi - Cookbook

By following these steps, you will be able to deploy rdo-manager using khaleesi on a CentOS machine with a basic configuration

Requirements

For deploying rdo-manager you will need at least a baremetal machine which must has the following minimum system requirements:

```
CentOS-7
Virtualization hardware extenstions enabled (nested KVM is not supported)
1 quad core CPU
12 GB free memory
120 GB disk space
```

Khaleesi driven RDO-Manager deployments only support the following operating systems:

```
CentOS 7 x86_64
RHEL 7.1 x86_64 ( Red Hat internal deployments only )
```

See the following documentation for system requirements:

```
http://docs.openstack.org/developer/tripleo-docs/environments/environments.html
↪#minimum-system-requirements
```

Note: There is an internal khaleesi-settings git repository that contains the settings and configuration for RHEL deployments. Do not attempt to use a RHEL bare metal host or RHEL options in ksgen using these instructions

Deploy rdo-manager

Installation:

Note: The following steps should be executed from the machine that will be operating Khaleesi, not the machine it will be installing the undercloud and overcloud on.

Get the code :

khaleesi on Github:

```
git clone git@github.com:redhat-openstack/khaleesi.git
```

khaleesi-settings on Github:

```
git clone git@github.com:redhat-openstack/khaleesi-settings.git
```

Install tools and system packages:

```
sudo yum install -y python-virtualenv gcc
```

or on Fedora 22:

```
sudo dnf install -y python-virtualenv gcc
```

Create the virtual environment, install ansible, and ksgen util:

```
virtualenv venv
source venv/bin/activate
pip install ansible==1.9.2
cd khaleesi/tools/ksgen
python setup.py install
cd ../..
```

Note: If you get a errors with kcli installation make sure you have all system development tools intalled on your local machine: python2-devel for Fedora CentOS

Configuration:

Create the appropriate ansible.cfg for khaleesi:

```
cp ansible.cfg.example ansible.cfg
touch ssh.config.ansible
echo "" >> ansible.cfg
echo "[ssh_connection]" >> ansible.cfg
echo "ssh_args = -F ssh.config.ansible" >> ansible.cfg
```

SSH Keys:

Note: We assume that you will named the key : ~/id_rsa and ~/id_rsa.pub

---

Ensure that your ~/.ssh/id_rsa.pub file is in /root/.ssh/authorized_keys file on the baremetal virt host:

```
ssh-copy-id root@<ip address of baremetal virt host>
```

Deployment Configuration:

Export the ip or fqdn hostname of the test box you will use as the virtual host for osp-director:

```
export TEST_MACHINE=<ip address of baremetal virt host>
```

Create a ksgen-settings file for Khaleesi to be able to get options and settings:

```
ksgen --config-dir settings generate \
    --provisioner=manual \
    --product=rdo \
    --product-version=liberty \
    --product-version-build=last_known_good \
    --product-version-repo=delorean_mgt \
    --distro=centos-7.0 \
    --installer=rdo_manager \
    --installer-env=virthost \
    --installer-images=import_rdo \
    --installer-network-isolation=none \
    --installer-network-variant=ml2-vxlan \
    --installer-post_action=none \
    --installer-topology=minimal \
    --installer-tempest=smoke \
    --workarounds=enabled \
    --extra-vars @../khaleesi-settings/hardware_environments/virt/network_configs/
↪none/hw_settings.yml \
    ksgen_settings.yml
```

Note: The "base_dir" key is defined by either where you execute ksgen from or by the $WORKSPACE environment variable. The base_dir value should point to the directory where khaleesi and khaleesi-settings have been cloned.

If you want to have more informations about the options used by ksgen launch:

```
ksgen --config-dir=../khaleesi-settings/settings help
```

Note: This output will give you all options available in ksgen tools, You can also check into *Usage* for more examples.

Once all of these steps have been completed you will have a ksgen-settings file containing all the settings needed for deployment. Khaleesi will load all of the variables from this YAML file.

Review the ksgen_settings.yml file:

Deployment Execution:

Run your intended deployment:

```
ansible-playbook -vv --extra-vars @ksgen_settings.yml -i local_hosts playbooks/full-
↪job-no-test.yml
```

Cleanup

After you finished your work, you can simply remove the created instances by:

```
ansible-playbook -vv --extra-vars @ksgen_settings.yml -i hosts playbooks/cleanup.yml
```

Building rpms

You can use khaleesi to build rpms for you.

If you want to test manually a rpm with a patch from gerrit you can use the khaleesi infrastructure to do that.

Setup Configuration:

What you will need:

Ansible 1.9 installed I would recommend on a virtualenv:

```
virtualenv foobar
source foobar/bin/activate
pip install ansible==1.9.2
```

`rdopkg` is what is going to do the heavy lifting

> https://github.com/redhat-openstack/rdopkg

There's a public repo for the up to date version that can be installed like this:

```
wget https://copr.fedoraproject.org/coprs/jruzicka/rdopkg/repo/epel-7/jruzicka-rdopkg-
↪epel-7.repo
sudo cp jruzicka-rdopkg-epel-7.repo /etc/yum.repos.d

yum install -y rdopkg
```

Newer fedora versions uses dnf instead of yum so for the last step use:

```
dnf install -y rdopkg
```

You will aslo need a `rhpkg` or a `fedpkg` those can be obtained from yum or dnf:

```
yum install -y rhpkg
```

or:

```
yum install -y fedpkg
```

Again for newer fedora versions replace yum for dnf:

```
dnf install -y rhpkg
dnf install -y fedpkg
```

In khaleesi will build the package locally (on a /tmp/tmp.patch_rpm_* directory) but in order to do that it needs a file called `hosts_local` on your khaleesi folder

The `hosts_local` should have this content:

```
[local]
localhost ansible_connection=local
```

ksgen_settings needed

Once you've got that you need to setup what gerrit patch you want to test:

```
export GERRIT_BRANCH=<the_branch>
export GERRIT_REFSPEC=<the_refspec>
export EXECUTOR_NUMBER=0; #needed for now
```

Then you'll need to load this structure into your `ksgen_settings.yml`:

```
patch:
  upstream:
    name: "upstream-<package>"
    url: "https://git.openstack.org/openstack/<package>"
  gerrit:
    name: "gerrit-<package>"
    url: "<gerrit_url>"
    branch: "{{ lookup('env', 'GERRIT_BRANCH') }}"
    refspec: "{{ lookup('env', 'GERRIT_REFSPEC') }}"
  dist_git:
    name: "openstack-<package>"
    url: "<dist-git_url>"
    use_director: False
```

There's two ways to do that:

Either set the values via extra-vars:

```
ksgen --config-dir settings \
  generate \
    --distro=rhel-7.1 \
    --product=rhos \
    --product-version=7.0
    --extra-vars patch.upstream.name=upstream-<package> \
    --extra-vars patch.upstream.url=https://git.openstack.org/openstack/<package> \
    --extra-vars patch.gerrit.name=gerrit-<package> \
    --extra-vars patch.gerrit.url=<gerrit_url> \
    --extra-vars patch.gerrit.branch=$GERRIT_BRANCH \
    --extra-vars patch.gerrit.refspec=$GERRIT_REFSPEC \
    --extra-vars patch.dist_git.name=openstack-<package> \
    --extra-vars patch.dist_git.url=<dist-git_url> \
    --extra-vars @../khaleesi-settings/settings/product/rhos/private_settings/redhat_
↪internal.yml \
    ksgen_settings.yml
```

Or if khaleesi already has the settings for package you are trying to build on khaleesi/settings/rpm/<package>.yml you can do this second method:

```
ksgen --config-dir settings \
  generate \
    --distro=rhel-7.1 \
    --product=rhos \
    --product-version=7.0
    --rpm=<package>
    --extra-vars @../khaleesi-settings/settings/product/rhos/private_settings/redhat_
↪internal.yml \
    ksgen_settings.yml
```

Note: At this time this second method works only for instack-undercloud, ironic, tripleo-heat-templates and python-rdomanager-oscplugin

---

Playbook usage

Then just call the playbook with that ksgen_settings:

```
ansible-playbook -vv --extra-vars @ksgen_settings.yml -i local_hosts playbooks/build_
↪gate_rpm.yml
```

When the playbook is done the generated rpms will be on the `generated_rpms` of your `khaleesi` directory

CHAPTER 12

Log collection in Khaleesi

Collecting and saving the logs after a job run from the affected machines is an important step. Khaleesi has a playbook dedicated for this process.

To collect the logs from the machines, run the `khaleesi/playbooks/collect_logs.yml` playbook right after the job run with the same settings and host file. The localhost and the machine called `host0` is excluded from the log collection. They are usually long running machines (the slave and if it exists, a virtual host) that have a large amount and mostly irrelevant logs.

What files are gathered?

Quite a few diagnostic commands are run on the machines (found in the playbook) and then a set of log files collected. If some specific setting used require specific logs to be collected, it's practical to add these files to that specific settings yaml:

```
job:
    archive:
        - /var/foo/*.log
        - /var/bar/engine.log
        - /opt/baz/*.xml
        ...
```

If any file is missing it won't cause the log collection to fail.

Methods of gathering logs

By default the logs are stored on the machine running ansible in `khaleesi/collected_files`, each machine's log in a different `tar.gz` file.

This behavior can be changed by the `--logs=gzip` option, which will result in individually gzipping the files instead.

Uploading the logs

When using the gzip method, it's possible to upload the logs on an artifact server. Ideally the logs are then exposed on a HTTP server for online browsing.

To set up a new site, add a new option to `khaleesi/settings/logs/gzip/site`, then use the `--logs-site` option when running ksgen.

An example site definition looks like:

```
job:
    rsync_logs: true
    rsync_path: myuser@example.com:/opt/artifacts
    artifact_url: http://example.com/artifacts
```

The `rsync_path` should be something that rsync understands as a destination, and the `artifact_url` will be used to generate the link to the logs. This method assumes the job runs on a Jenkins server, so the `$BUILD_TAG` variable should be set in the environment.

After the upload, the logs are deleted from the local machine and a link file is created as `khaleesi/collected_files/full_logs.html`. This file should be added as an artifact to the Jenkins job definition and can be used as a one click redirect to the job specific artifacts.

Setting up an artifact storage

The machine running the job should be able to upload the logs without a password to the artifact server. When using Apache to expose the logs for browsing, the following httpd settings will allow transparent browsing of the gzipped files:

```
Alias /artifacts /opt/artifacts

<Directory /opt/artifacts>
    Options +Indexes
    RewriteCond   %{HTTP:Accept-Encoding} gzip
    RewriteCond   %{LA-U:REQUEST_FILENAME}.gz -f
    RewriteRule   ^(.+)$ $1.gz [L]
    <FilesMatch ".*\.gz$">
        ForceType text/plain
        AddDefaultCharset UTF-8
        AddEncoding x-gzip gz
    </FilesMatch>
</Directory>
```

It's recommended to put these settings to a separate file in the `/etc/httpd/conf.d` directory.

Pruning old logs

The artifact directory could grow too big over time, thus it's useful to set up a cron job for deleting obsolete logs.

On the artifact server, add a line to /etc/crontab similar to this:

```
0 0 * * * rhos-ci find /opt/artifacts -maxdepth 1 -type d -ctime +14 -exec rm -rf {}␣
↪+;
```

This will delete any artifact directory in /opt/artifacts that is older than 14 days. It's useful to match the Jenkins artifact retention time with the time specified here to avoid broken links in Jenkins.

Use in Jenkins job definitions

To use the advanced gzip+upload method, modify your jobs the following way:

Add `--logs=gzip --logs-site=mysite` to the ksgen invocation of your builder, for example:

```
ksgen --config-dir settings generate \
    --provisioner=manual \
    ...
    --logs=gzip \
    --logs-site=downstream \
    ...
    ksgen_settings.yml
```

Add `**/full_logs.html` to the list of artifacts:

```
- publisher:
    name: default-publishers
    publishers:
        - archive:
            artifacts: '**/collect_logs.txt, **/cleanup.txt, **/nosetests.xml, **/
↪ksgen_settings.yml, **/full_logs.html'
```

After regnerating the jobs, the logs should start appearing on the artifact server.

It's practical to match the Jenkins artifact retention time with the artifact server retention time to avoid broken links in Jenkins:

```
- defaults:
    name: job-defaults
    ...
    logrotate:
        daysToKeep: 14
        artifactDaysToKeep: 14
    ...
```

CHAPTER 13

Indices and tables

- genindex
- modindex
- search