kayobe Documentation

Release 9.4.1.dev11

OpenStack Foundation

Oct 14, 2022

CONTENTS

1 Overview **1**

2 Kayobe **3**
2.1 Features 4

3 Contents **5**
3.1 Getting Started 5
3.2 Architecture 5
3.3 Support Matrix 7
3.4 Installation 7
3.5 Usage 10
3.6 Configuration Guide 11
3.7 Deployment 97
3.8 Upgrading 106
3.9 Administration 114
3.10 Resources 126
3.11 Advanced Documentation 126
3.12 Contributor Guide 134

CHAPTER

ONE

OVERVIEW

Welcome to the Kayobe documentation, the official source of information for understanding and using Kayobe.

This documentation is maintained at opendev.org here. Feedback and contributions welcome, see *contributing* for information on how.

CHAPTER

TWO

KAYOBE

Kayobe enables deployment of containerised OpenStack to bare metal.

Containers offer a compelling solution for isolating OpenStack services, but running the control plane on an orchestrator such as Kubernetes or Docker Swarm adds significant complexity and operational overheads.

The hosts in an OpenStack control plane must somehow be provisioned, but deploying a secondary OpenStack cloud to do this seems like overkill.

Kayobe stands on the shoulders of giants:

- OpenStack bifrost discovers and provisions the cloud
- OpenStack kolla builds container images for OpenStack services
- OpenStack kolla-ansible delivers painless deployment and upgrade of containerised OpenStack services

To this solid base, kayobe adds:

- Configuration of cloud host OS & flexible networking
- Management of physical network devices
- A friendly openstack-like CLI

All this and more, automated from top to bottom using Ansible.

- Free software: Apache license
- Documentation: https://docs.openstack.org/kayobe/latest/
- Source: https://opendev.org/openstack/kayobe
- Bugs: https://storyboard.openstack.org/#!/project/openstack/kayobe
- Release Notes: https://docs.openstack.org/releasenotes/kayobe/
- IRC: #openstack-kolla on OFTC

2.1 Features

- Heavily automated using Ansible
- *kayobe* Command Line Interface (CLI) for cloud operators
- Deployment of a *seed* VM used to manage the OpenStack control plane
- Configuration of physical network infrastructure
- Discovery, introspection and provisioning of control plane hardware using OpenStack bifrost
- Deployment of an OpenStack control plane using OpenStack kolla-ansible
- Discovery, introspection and provisioning of bare metal compute hosts using OpenStack ironic and ironic inspector
- Virtualised compute using OpenStack nova
- Containerised workloads on bare metal using OpenStack magnum
- Big data on bare metal using OpenStack sahara
- Control plane and workload monitoring and log aggregation using OpenStack monasca

CHAPTER

THREE

CONTENTS

3.1 Getting Started

We advise new users start by reading the *Architecture* documentation first in order to understand Kayobes various components.

For users wishing to learn interactively we recommend starting at either the *all-in-one overcloud* deployment or the *A Universe From Nothing* deployment guide.

Once familiar with Kayobes constituent parts, move on to the *Installation* section to prepare a baremetal environment and then *Deployment* to deploy to it.

- *Architecture* - The function of Kayobes host and networking components
- *Installation* - The prerequisites and options for installing Kayobe
- *Usage* - An introduction to the Kayobe CLI
- *Configuration* - How to configure Kayobes various components
- *Deployment*- Using Kayobe to deploy OpenStack
- *Upgrading* - Upgrading from one OpenStack release to another
- *Administration* - Post-deploy administration tasks
- *Resources* - External links to Kayobe resources
- *Contributor* - Contributing to Kayobe and deploying Kayobe development environments

3.2 Architecture

3.2.1 Hosts in the System

In a system deployed by Kayobe we define a number of classes of hosts.

Ansible control host The Ansible control host is the host on which kayobe, kolla and kolla-ansible will be installed, and is typically where the cloud will be managed from.

Seed host The seed host runs the bifrost deploy container and is used to provision the cloud hosts. By default, container images are built on the seed. Typically the seed host is deployed as a VM but this is not mandatory.

Cloud hosts The cloud hosts run the OpenStack control plane, network, monitoring, storage, and virtualised compute services. Typically the cloud hosts run on bare metal but this is not mandatory.

Bare metal compute hosts In a cloud providing bare metal compute services to tenants via ironic, these hosts will run the bare metal tenant workloads. In a cloud with only virtualised compute this category of hosts does not exist.

Note: In many cases the control and seed host will be the same, although this is not mandatory.

Cloud Hosts

Cloud hosts can further be divided into subclasses.

Controllers Controller hosts run the OpenStack control plane services.

Network Network hosts run the neutron networking services and load balancers for the OpenStack API services.

Monitoring Monitoring host run the control plane and workload monitoring services. Currently, kayobe does not deploy any services onto monitoring hosts.

Virtualised compute hypervisors Virtualised compute hypervisors run the tenant Virtual Machines (VMs) and associated OpenStack services for compute, networking and storage.

3.2.2 Networks

Kayobes network configuration is very flexible but does define a few default classes of networks. These are logical networks and may map to one or more physical networks in the system.

Overcloud out-of-band network Name of the network used by the seed to access the out-of-band management controllers of the bare metal overcloud hosts.

Overcloud provisioning network The overcloud provisioning network is used by the seed host to provision the cloud hosts.

Workload out-of-band network Name of the network used by the overcloud hosts to access the out-of-band management controllers of the bare metal workload hosts.

Workload provisioning network The workload provisioning network is used by the cloud hosts to provision the bare metal compute hosts.

Internal network The internal network hosts the internal and admin OpenStack API endpoints.

Public network The public network hosts the public OpenStack API endpoints.

External network The external network provides external network access for the hosts in the system.

3.3 Support Matrix

3.3.1 Supported Operating Systems

Kayobe supports the following host Operating Systems (OS):

- CentOS 8

Note: CentOS 7 is no longer supported as a host OS. The Train release supports both CentOS 7 and 8, and provides a route for migration. See the Kayobe Train documentation for information on migrating to CentOS 8.

3.3.2 Supported container images

For details of container image distributions supported by Kolla Ansible, see the support matrix.

For details of which images are supported on which distributions, see the Kolla support matrix.

3.4 Installation

Kayobe can be installed via the released Python packages on PyPI, or from source. Installing from PyPI ensures the use of well used and tested software, whereas installing from source allows for the use of unreleased or patched code. Installing from a Python package is supported from Kayobe 5.0.0 onwards.

3.4.1 Prerequisites

Currently Kayobe supports the following Operating Systems on the Ansible control host:

- CentOS 8
- Ubuntu 16.04

See the *support matrix* for details of supported Operating Systems for other hosts.

To avoid conflicts with python packages installed by the system package manager it is recommended to install Kayobe in a virtualenv. Ensure that the `virtualenv` python module is available on the Ansible control host. It is necessary to install the GCC compiler chain in order to build the extensions of some of kayobes python dependencies.

On CentOS:

```
$ dnf install -y python3-devel python3-virtualenv gcc libffi-devel
```

On Ubuntu:

```
$ apt install -y python3-dev python3-virtualenv gcc libffi-dev
```

If installing Kayobe from source, then Git is required for cloning and working with the source code repository.

On CentOS:

```
$ dnf install -y git
```

On Ubuntu:

```
$ apt install -y git
```

3.4.2 Local directory structure

The directory structure for a Kayobe Ansible control host environment is configurable, but the following is recommended, where `<base_path>` is the path to a top level directory:

```
<base_path>/
    src/
        kayobe/
        kayobe-config/
        kolla-ansible/
    venvs/
        kayobe/
        kolla-ansible/
```

This pattern ensures that all dependencies for a particular environment are installed under a single top level path, and nothing is installed to a shared location. This allows for the option of using multiple Kayobe environments on the same control host.

Creation of a `kayobe-config` source code repository will be covered in the *configuration guide*. The Kolla Ansible source code checkout and Python virtual environment will be created automatically by kayobe.

Not all of these directories will be used in all scenarios - if Kayobe or Kolla Ansible are installed from a Python package then the source code repository is not required.

3.4.3 Installation from PyPI

This section describes how to install Kayobe from a Python package in a virtualenv. This is supported from Kayobe 5.0.0 onwards.

First, change to the top level directory, and make the directories for source code repositories and python virtual environments:

```
$ cd <base_path>
$ mkdir -p src venvs
```

Create a virtualenv for Kayobe:

```
$ virtualenv <base_path>/venvs/kayobe
```

Activate the virtualenv and update pip:

```
$ source <base_path>/venvs/kayobe/bin/activate
(kayobe) $ pip install -U pip
```

If using the latest version of Kayobe:

```
(kayobe) $ pip install kayobe
```

Alternatively, to install a specific release of Kayobe:

```
(kayobe) $ pip install kayobe==5.0.0
```

Finally, deactivate the virtualenv:

```
(kayobe) $ deactivate
```

3.4.4 Installation from source

This section describes how to install Kayobe from source in a virtualenv.

First, change to the top level directory, and make the directories for source code repositories and python virtual environments:

```
$ cd <base_path>
$ mkdir -p src venvs
```

Next, obtain the Kayobe source code. For example:

```
$ cd <base_path>/src
$ git clone https://opendev.org/openstack/kayobe.git -b stable/
↪victoria
```

Create a virtualenv for Kayobe:

```
$ virtualenv <base_path>/venvs/kayobe
```

Activate the virtualenv and update pip:

```
$ source <base_path>/venvs/kayobe/bin/activate
(kayobe) $ pip install -U pip
```

Install Kayobe and its dependencies using the source code checkout:

```
(kayobe) $ cd <base_path>/src/kayobe
(kayobe) $ pip install .
```

Finally, deactivate the virtualenv:

```
(kayobe) $ deactivate
```

Editable source installation

From Kayobe 5.0.0 onwards it is possible to create an editable install of Kayobe. In an editable install, any changes to the Kayobe source tree will immediately be visible when running any Kayobe commands. To create an editable install, add the `-e` flag:

```
(kayobe) $ cd <base_path>/src/kayobe
(kayobe) $ pip install -e .
```

This is particularly useful when installing Kayobe for development.

3.5 Usage

3.5.1 Command Line Interface

Note: Where a prompt starts with `(kayobe)` it is implied that the user has activated the Kayobe virtualenv. This can be done as follows:

```
$ source /path/to/venv/bin/activate
```

To deactivate the virtualenv:

```
(kayobe) $ deactivate
```

To see information on how to use the `kayobe` CLI and the commands it provides:

```
(kayobe) $ kayobe help
```

As the `kayobe` CLI is based on the `cliff` package (as used by the `openstack` client), it supports tab auto-completion of subcommands. This can be activated by generating and then sourcing the bash completion script:

```
(kayobe) $ kayobe complete > kayobe-complete
(kayobe) $ source kayobe-complete
```

Working with Ansible Vault

If Ansible vault has been used to encrypt Kayobe configuration files, it will be necessary to provide the `kayobe` command with access to vault password. There are three options for doing this:

Prompt Use `kayobe --ask-vault-pass` to prompt for the password.

File Use `kayobe --vault-password-file <file>` to read the password from a (plain text) file.

Environment variable Export the environment variable `KAYOBE_VAULT_PASSWORD` to read the password from the environment.

Limiting Hosts

Sometimes it may be necessary to limit execution of kayobe or kolla-ansible plays to a subset of the hosts. The `--limit <SUBSET>` argument allows the kayobe ansible hosts to be limited. The `--kolla-limit <SUBSET>` argument allows the kolla-ansible hosts to be limited. These two options may be combined in a single command. In both cases, the argument provided should be an Ansible host pattern, and will ultimately be passed to `ansible-playbook` as a `--limit` argument.

Tags

Ansible tags provide a useful mechanism for executing a subset of the plays or tasks in a playbook. The `--tags <TAGS>` argument allows execution of kayobe ansible playbooks to be limited to matching plays and tasks. The `--kolla-tags <TAGS>` argument allows execution of kolla-ansible ansible playbooks to be limited to matching plays and tasks. The `--skip-tags <TAGS>` and `--kolla-skip-tags <TAGS>` arguments allow for avoiding execution of matching plays and tasks.

3.6 Configuration Guide

The configuration guide is split into two parts - scenarios and reference. The scenarios section provides information on configuring Kayobe for different scenarios. The reference section provides detailed information on many of Kayobes configuration options.

3.6.1 Configuration Scenarios

This section provides information on configuring Kayobe for different scenarios.

All in one scenario

Note: This documentation is intended as a walk through of the configuration required for a minimal all-in-one overcloud host. If you are looking for an all-in-one environment for test or development, see *Automated Setup*.

This scenario describes how to configure an all-in-one controller and compute node using Kayobe. This is a very minimal setup, and not one that is recommended for a production environment, but is useful for learning about how to use and configure Kayobe.

Prerequisites

This scenario requires a basic understanding of Linux, networking and OpenStack.

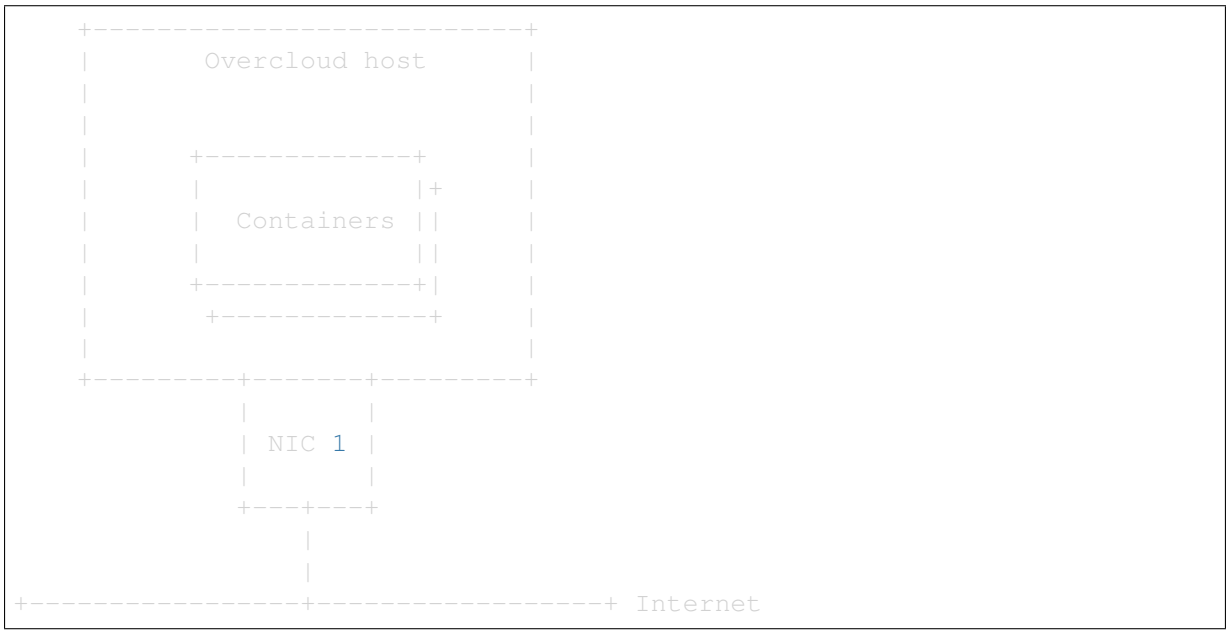
It also requires a single host running a *supported operating system* (VM or bare metal), with:

- 1 CPU
- 8GB RAM
- 40GB disk
- at least one network interface that has Internet access

You will need access to a user account with passwordless sudo. The default user in a cloud image (e.g. `centos` or `ubuntu`) is typically sufficient. This user will be used to run Kayobe commands. It will also be used by Kayobe to bootstrap other user accounts.

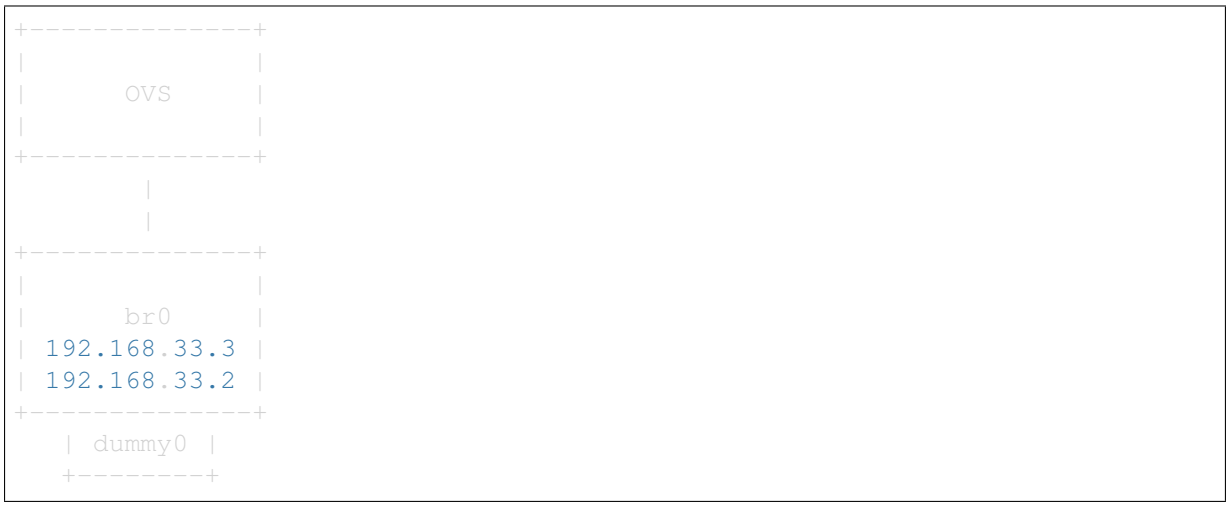
Overview

An all in one environment consists of a single node that provides both control and compute services. There is no seed host, and no provisioning of the overcloud host. Customisation is minimal, in order to demonstrate the basic required configuration in Kayobe:



The networking in particular is relatively simple. The main interface of the overcloud host, labelled NIC 1 in the above diagram, will be used only for connectivity to the host and Internet access. A single Kayobe network called `aio` carries all control plane traffic, and is based on virtual networking that is local to the host.

Later in this tutorial, we will create a dummy interface called `dummy0`, and plug it into a bridge called `br0`:



The use of a bridge here allows Kayobe to connect this network to the Open vSwitch network, while maintaining an IP address on the bridge. Ordinarily, `dummy0` would be a NIC providing connectivity to a physical network. Were using a dummy interface here to keep things simple by using a fixed IP subnet, `192.168.33.0/24`. The bridge will be assigned a static IP address of `192.168.33.3`, and this address will by used for various things, including Ansible SSH access and OpenStack control plane traffic. Kolla Ansible will manage a Virtual IP (VIP) address of `192.168.33.2` on `br0`, which will

be used for OpenStack API endpoints.

Contents

Overcloud

Note: This documentation is intended as a walk through of the configuration required for a minimal all-in-one overcloud host. If you are looking for an all-in-one environment for test or development, see *Automated Setup*.

Preparation

Use the bootstrap user described in *prerequisites* to access the machine.

As described in the *overview*, we will use a bridge (`br0`) and a dummy interface (`dummy0`) for control plane networking. Use the following commands to create them and assign the bridge a static IP address of `192.168.33.3`:

```
sudo ip l add br0 type bridge
sudo ip l set br0 up
sudo ip a add 192.168.33.3/24 dev br0
sudo ip l add dummy0 type dummy
sudo ip l set dummy0 up
sudo ip l set dummy0 master br0
```

This configuration is not persistent, and must be recreated if the VM is rebooted.

Installation

Follow the instructions in *Installation* to set up an Ansible control host environment. Typically this would be on a separate machine, but here we are keeping things as simple as possible.

Configuration

Clone the kayobe-config git repository, using the correct branch for the release you are deploying. In this example we will use the stable/victoria branch.

```
cd <base path>/src
git clone https://opendev.org/openstack/kayobe-config.git -b stable/
↪victoria
cd kayobe-config
```

This repository is bare, and needs to be populated. The repository includes an example inventory, which should be removed:

```
git rm etc/kayobe/inventory/hosts.example
```

Create an Ansible inventory file and add the machine to it. In this example our machine is called `controller0`. Since this is an all-in-one environment, we add the controller to the `compute` group, however normally dedicated compute nodes would be used.

Listing 1: `etc/kayobe/inventory/hosts`

```
# This host acts as the configuration management Ansible control host.␣
↪This must be
# localhost.
localhost ansible_connection=local

[controllers]
controller0

[compute:children]
controllers
```

The `inventory` directory also contains group variables for network interface configuration. In this example we will assume that the machine has a single network interface called `dummy0`. We will create a bridge called `br0` and plug `dummy0` into it. Replace the network interface configuration for the `controllers` group with the following:

Listing 2: `etc/kayobe/inventory/group_vars/controllers/network-interfaces`

```
# Controller interface on all-in-one network.
aio_interface: br0

# Interface dummy0 is plugged into the all-in-one network bridge.
aio_bridge_ports:
  - dummy0
```

In this scenario a single network called `aio` is used. We must therefore set the name of the default controller networks to `aio`:

Listing 3: `etc/kayobe/networks.yml`

```
---
# Kayobe network configuration.

############################################################################
↪####
# Network role to network mappings.

# Map all networks to the all-in-one network.

# Name of the network used for admin access to the overcloud
#admin_oc_net_name:
admin_oc_net_name: aio

# Name of the network used by the seed to manage the bare metal overcloud
# hosts via their out-of-band management controllers.
#oob_oc_net_name:

# Name of the network used by the seed to provision the bare metal␣
↪overcloud
```

(continues on next page)

(continued from previous page)

```
# hosts.
#provision_oc_net_name:

# Name of the network used by the overcloud hosts to manage the bare metal
# compute hosts via their out-of-band management controllers.
#oob_wl_net_name:

# Name of the network used by the overcloud hosts to provision the bare␣
↪metal
# workload hosts.
#provision_wl_net_name:

# Name of the network used to expose the internal OpenStack API endpoints.
#internal_net_name:
internal_net_name: aio

# List of names of networks used to provide external network access via
# Neutron.
# Deprecated name: external_net_name
# If external_net_name is defined, external_net_names will default to a␣
↪list
# containing one item, external_net_name.
#external_net_names:
external_net_names:
  - aio

# Name of the network used to expose the public OpenStack API endpoints.
#public_net_name:
public_net_name: aio

# Name of the network used by Neutron to carry tenant overlay network␣
↪traffic.
#tunnel_net_name:
tunnel_net_name: aio

# Name of the network used to carry storage data traffic.
#storage_net_name:
storage_net_name: aio

# Name of the network used to carry storage management traffic.
#storage_mgmt_net_name:
storage_mgmt_net_name: aio

# Name of the network used to carry swift storage data traffic.
#swift_storage_net_name:

# Name of the network used to carry swift storage replication traffic.
#swift_storage_replication_net_name:

# Name of the network used to perform hardware introspection on the bare␣
↪metal
# workload hosts.
#inspection_net_name:

# Name of the network used to perform cleaning on the bare metal workload
# hosts
```

(continues on next page)

(continued from previous page)

```
#cleaning_net_name:

################################################################################
↪####
# Network definitions.

<omitted for clarity>
```

Next the `aio` network must be defined. This is done using the various attributes described in *Network Configuration*. These values should be adjusted to match the environment. The `aio_vip_address` variable should be a free IP address in the same subnet for the virtual IP address of the OpenStack API.

Listing 4: `etc/kayobe/networks.yml`

```
<omitted for clarity>

################################################################################
↪####
# Network definitions.

# All-in-one network.
aio_cidr: 192.168.33.0/24
aio_vip_address: 192.168.33.2

################################################################################
↪####
# Network virtual patch link configuration.

<omitted for clarity>
```

Kayobe will automatically allocate IP addresses. In this case however, we want to ensure that the host uses the same IP address it has currently, to avoid loss of connectivity. We can do this by populating the network allocation file. Use the correct hostname and IP address for your environment.

Listing 5: `etc/kayobe/network-allocation.yml`

```
---
aio_ips:
  controller0: 192.168.33.3
```

Kayobe uses a bootstrap user to create a `stack` user account. By default, this user is `centos` on CentOS, and `ubuntu` on Ubuntu, in line with the default user in the official cloud images. If you are using a different bootstrap user, set the `controller_bootstrap_user` variable in `etc/kayobe/controllers.yml`. For example, to set it to `cloud-user` (as seen in MAAS):

Listing 6: `etc/kayobe/controllers.yml`

```
controller_bootstrap_user: "cloud-user"
```

By default, on systems with SELinux enabled, Kayobe will disable SELinux and reboot the system to apply the change. In a test or development environment this can be a bit disruptive, particularly when using ephemeral network configuration. To avoid rebooting the system after disabling SELinux, set `disable_selinux_do_reboot` to `false` in `etc/kayobe/globals.yml`.

Listing 7: `etc/kayobe/globals.yml`

```
disable_selinux_do_reboot: false
```

In a development environment, we may wish to tune some Kolla Ansible variables. Using QEMU as the virtualisation type will be necessary if KVM is not available. Reducing the number of OpenStack service workers helps to avoid using too much memory.

Listing 8: `etc/kayobe/kolla/globals.yml`

```
# Most development environments will use nested virtualisation, and we can
↪'t
# guarantee that nested KVM support is available. Use QEMU as a lowest␣
↪common
# denominator.
nova_compute_virt_type: qemu

# Reduce the control plane's memory footprint by limiting the number of␣
↪worker
# processes to one per-service.
openstack_service_workers: "1"
```

We can see the changes that have been made to the configuration.

```
cd <base path>/src/kayobe-config
git status

On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    deleted:    etc/kayobe/inventory/hosts.example

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   etc/kayobe/globals.yml
    modified:   etc/kayobe/inventory/group_vars/controllers/network-
↪interfaces
    modified:   etc/kayobe/kolla/globals.yml
    modified:   etc/kayobe/networks.yml

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    etc/kayobe/inventory/hosts
    etc/kayobe/network-allocation.yml
```

The `git diff` command is also helpful. Once all configuration changes have been made, they should be committed to the kayobe-config git repository.

```
cd <base path>/src/kayobe-config
git add etc/kayobe/inventory/hosts etc/kayobe/network-allocation.yml
git add --update
git commit -m "All in one scenario config"
```

In a real environment these changes would be pushed to a central repository.

Deployment

We are now ready to perform a deployment.

Activate the Kayobe virtual environment:

```
cd <base path>/venvs/kayobe
source bin/activate
```

Activate the Kayobe configuration environment:

```
cd <base path>/src/kayobe-config
source kayobe-env
```

Bootstrap the control host:

```
kayobe control host bootstrap
```

Configure the overcloud host:

```
kayobe overcloud host configure
```

After this command has run, some files in the kayobe-config repository will have changed. Kayobe performs static allocation of IP addresses, and tracks them in `etc/kayobe/network-allocation.yml`. Normally there may be changes to this file, but in this case we manually added the IP address of `controller0` earlier. Kayobe uses tools provided by Kolla Ansible to generate passwords, and stores them in `etc/kayobe/kolla/passwords.yml`. It is important to track changes to this file.

```
cd <base path>/src/kayobe-config
git add etc/kayobe/kolla/passwords.yml
git commit -m "Add autogenerated passwords for Kolla Ansible"
```

Pull overcloud container images:

```
kayobe overcloud container image pull
```

Deploy overcloud services:

```
kayobe overcloud service deploy
```

Testing

The `init-runonce` script provided by Kolla Ansible (not for production) can be used to setup some resources for testing. This includes:

- some flavors
- a cirros image
- an external network
- a tenant network and router

- security group rules for ICMP, SSH, and TCP ports 8000 and 8080
- an SSH key
- increased quotas

For the external network, use the same subnet as before, with an allocation pool range containing free IP addresses:

```
pip install python-openstackclient
export EXT_NET_CIDR=192.168.33.0/24
export EXT_NET_GATEWAY=192.168.33.3
export EXT_NET_RANGE="start=192.168.33.4,end=192.168.33.254"
source "${KOLLA_CONFIG_PATH:-/etc/kolla}/admin-openrc.sh"
${KOLLA_SOURCE_PATH}/tools/init-runonce
```

Create a server instance, assign a floating IP address, and check that it is accessible.

```
openstack server create --image cirros --flavor m1.tiny --key-name mykey --
↪network demo-net demo1
openstack floating ip create public1
```

The floating IP address is displayed after it is created, in this example it is `192.168.33.4`:

```
openstack server add floating ip demo1 192.168.33.4
ssh cirros@192.168.33.4
```

3.6.2 Configuration Reference

This section provides detailed information on many of Kayobes configuration options.

Kayobe Configuration

This section covers configuration of Kayobe. As an Ansible-based project, Kayobe is for the most part configured using YAML files.

Configuration Location

Kayobe configuration is by default located in `/etc/kayobe` on the Ansible control host. This location can be overridden to a different location to avoid touching the system configuration directory by setting the environment variable `KAYOBE_CONFIG_PATH`. Similarly, kolla configuration on the Ansible control host will by default be located in `/etc/kolla` and can be overridden via `KOLLA_CONFIG_PATH`.

Configuration Directory Layout

The Kayobe configuration directory contains Ansible `extra-vars` files and the Ansible inventory. An example of the directory structure is as follows:

```
extra-vars1.yml
extra-vars2.yml
inventory/
    group_vars/
        group1-vars
        group2-vars
    groups
    host_vars/
        host1-vars
        host2-vars
    hosts
```

Configuration Patterns

Ansibles variable precedence rules are fairly well documented and provide a mechanism we can use for providing site localisation and customisation of OpenStack in combination with some reasonable default values. For global configuration options, Kayobe typically uses the following patterns:

- Playbook group variables for the *all* group in `<kayobe repo>/ansible/group_vars/all/*` set **global defaults**. These files should not be modified.
- Playbook group variables for other groups in `<kayobe repo>/ansible/group_vars/<group>/*` set **defaults for some subsets of hosts**. These files should not be modified.
- Extra-vars files in `${KAYOBE_CONFIG_PATH}/*.yml` set **custom values for global variables** and should be used to apply global site localisation and customisation. By default these variables are commented out.

Additionally, variables can be set on a per-host basis using inventory host variables files in `${KAYOBE_CONFIG_PATH}/inventory/host_vars/*`. It should be noted that variables set in extra-vars files take precedence over per-host variables.

Configuring Kayobe

The kayobe-config git repository contains a Kayobe configuration directory structure and unmodified configuration files. This repository can be used as a mechanism for version controlling Kayobe configuration. As Kayobe is updated, the configuration should be merged to incorporate any upstream changes with local modifications.

Alternatively, the baseline Kayobe configuration may be copied from a checkout of the Kayobe repository to the Kayobe configuration path:

```
$ mkdir -p ${KAYOBE_CONFIG_PATH:-/etc/kayobe/}
$ cp -r etc/kayobe/* ${KAYOBE_CONFIG_PATH:-/etc/kayobe/}
```

Once in place, each of the YAML and inventory files should be manually inspected and configured as required.

Inventory

The inventory should contain the following hosts:

Ansible Control host This should be localhost.

Seed hypervisor If provisioning a seed VM, a host should exist for the hypervisor that will run the VM, and should be a member of the `seed-hypervisor` group.

Seed The seed host, whether provisioned as a VM by Kayobe or externally managed, should exist in the `seed` group.

Cloud hosts and bare metal compute hosts are not required to exist in the inventory if discovery of the control plane hardware is planned, although entries for groups may still be required.

Use of advanced control planes with multiple server roles and customised service placement across those servers is covered in *Control Plane Service Placement*.

Site Localisation and Customisation

Site localisation and customisation is applied using Ansible extra-vars files in `${KAYOBE_CONFIG_PATH}/*.yml`.

Configuration of Ansible

Ansible configuration is described in detail in the Ansible documentation. In addition to the standard locations, Kayobe supports using an Ansible configuration file located in the Kayobe configuration at `${KAYOBE_CONFIG_PATH}/ansible.cfg`. Note that if the `ANSIBLE_CONFIG` environment variable is specified it takes precedence over this file.

Encryption of Secrets

Kayobe supports the use of Ansible vault to encrypt sensitive information in its configuration. The `ansible-vault` tool should be used to manage individual files for which encryption is required. Any of the configuration files may be encrypted. Since encryption can make working with Kayobe difficult, it is recommended to follow best practice, adding a layer of indirection and using encryption only where necessary.

Location of data files

Kayobe needs to know where to find any files not contained within its python package; this includes its Ansible playbooks and any other files it needs for runtime operation. These files are known collectively as data files.

Kayobe will attempt to detect the location of its data files automatically. However, if you have installed kayobe to a non-standard location this auto-detection may fail. It is possible to manually override the path using the environment variable: `KAYOBE_DATA_FILES_PATH`. This should be set to a path with the following structure:

```
requirements.yml
ansible/
    roles/
        ...
    ...
```

Where `ansible` is the `ansible` directory from the source checkout and `...` is an elided representation of any files and subdirectories contained within that directory.

Ansible

Ansible configuration is described in detail in the Ansible documentation. It is explained elsewhere in this guide how to configure Ansible for *Kayobe* and *Kolla Ansible*.

In this section we cover some options for tuning Ansible for performance and scale.

SSH pipelining

SSH pipelining is disabled in Ansible by default, but is generally safe to enable, and provides a reasonable performance improvement.

Listing 9: `$KAYOBE_CONFIG_PATH/ansible.cfg`

```
[ssh_connection]
pipelining = True
```

Forks

By default Ansible executes tasks using a fairly conservative 5 process forks. This limits the parallelism that allows Ansible to scale. Most Ansible control hosts will be able to handle far more forks than this. You will need to experiment to find out the CPU, memory and IO limits of your machine.

For example, to increase the number of forks to 20:

Listing 10: `$KAYOBE_CONFIG_PATH/ansible.cfg`

```
[defaults]
forks = 20
```

Fact caching

Note: Fact caching will not work correctly in Kayobe prior to the Ussuri release.

By default, Ansible gathers facts for each host at the beginning of every play, unless `gather_facts` is set to `false`. With a large number of hosts this can result in a significant amount of time spent gathering facts.

One way to improve this is through Ansibles support for fact caching. In order to make this work with Kayobe, it is necessary to change Ansibles gathering configuration option to `smart`. Additionally, it is necessary to use separate fact caches for Kayobe and Kolla Ansible due to some of the facts (e.g. `ansible_facts.user_uid` and `ansible_facts.python`) differing.

Example

In the following example we configure Kayobe and Kolla Ansible to use fact caching using the jsonfile cache plugin.

Listing 11: `$KAYOBE_CONFIG_PATH/ansible.cfg`

```
[defaults]
gathering = smart
fact_caching = jsonfile
fact_caching_connection = /tmp/kayobe-facts
```

Listing 12: `$KAYOBE_CONFIG_PATH/kolla/ansible.cfg`

```
[defaults]
gathering = smart
fact_caching = jsonfile
fact_caching_connection = /tmp/kolla-ansible-facts
```

You may also wish to set the expiration timeout for the cache via `[defaults] fact_caching_timeout`.

Fact gathering

Fact filtering

Filtering of facts can be used to speed up Ansible. Environments with many network interfaces on the network and compute nodes can experience very slow processing with Kayobe and Kolla Ansible. This happens due to the processing of the large per-interface facts with each task. To avoid storing certain facts, we can use the `kayobe_ansible_setup_filter` variable, which is used as the `filter` argument to the `setup` module.

One case where this is particularly useful is to avoid collecting facts for virtual tap (beginning with t) and bridge (beginning with q) interfaces created by Neutron. These facts are large map values which can consume a lot of resources on the Ansible control host. Kayobe and Kolla Ansible typically do not need to reference them, so they may be filtered. For example, to avoid collecting facts beginning with q or t:

Listing 13: `$KAYOBE_CONFIG_PATH/globals.yml`

```
kayobe_ansible_setup_filter: "ansible_[!qt]*"
```

Similarly, for Kolla Ansible (notice the similar but different file names):

Listing 14: `$KAYOBE_CONFIG_PATH/kolla/globals.yml`

```
kolla_ansible_setup_filter: "ansible_[!qt]*"
```

This causes Ansible to collect but not store facts matching that pattern, which includes the virtual interface facts. Currently we are not referencing other facts matching the pattern within Kolla Ansible. Note that including the *ansible* prefix causes meta facts `module_setup` and `gather_subset` to be filtered, but this seems to be the only way to get a good match on the interface facts.

The exact improvement will vary, but has been reported to be as large as 18x on systems with many virtual interfaces.

Fact gathering subsets

It is also possible to configure which subsets of facts are gathered, via `kayobe_ansible_setup_gather_subset`, which is used as the `gather_subset` argument to the `setup` module. For example, if one wants to avoid collecting facts via facter:

Listing 15: `$KAYOBE_CONFIG_PATH/globals.yml`

```
kayobe_ansible_setup_gather_subset: "all,!facter"
```

Similarly, for Kolla Ansible (notice the similar but different file names):

Listing 16: `$KAYOBE_CONFIG_PATH/kolla/globals.yml`

```
kolla_ansible_setup_gather_subset: "all,!facter"
```

Physical Network Configuration

Kayobe supports configuration of physical network devices. This feature is optional, and this section may be skipped if network device configuration will be managed via other means.

Devices are added to the Ansible inventory, and configured using Ansibles networking modules. Configuration is applied via the `kayobe physical network configure` command. See *Physical Network* for details.

The following switch operating systems are currently supported:

- Cumulus Linux (via Network Command Line Utility (NCLU))
- Dell OS 6
- Dell OS 9
- Dell PowerConnect
- Juniper Junos OS
- Mellanox MLNX OS

Adding Devices to the Inventory

Network devices should be added to the Kayobe Ansible inventory, and should be members of the `switches` group.

Listing 17: `inventory/hosts`

```
[switches]
switch0
switch1
```

In some cases it may be useful to differentiate different types of switches, For example, a `mgmt` network might carry out-of-band management traffic, and a `ctl` network might carry control plane traffic. A group could be created for each of these networks, with each group being a child of the `switches` group.

Listing 18: `inventory/hosts`

```
[switches:children]
mgmt-switches
ctl-switches

[mgmt-switches]
switch0

[ctl-switches]
switch1
```

Network Device Configuration

Configuration is typically specific to each network device. It is therefore usually best to add a `host_vars` file to the inventory for each device. Common configuration for network devices can be added in a `group_vars` file for the `switches` group or one of its child groups.

Listing 19: `inventory/host_vars/switch0`

```
---
# Host configuration for switch0
ansible_host: 1.2.3.4
```

Listing 20: `inventory/host_vars/switch1`

```
---
# Host configuration for switch1
ansible_host: 1.2.3.5
```

Listing 21: `inventory/group_vars/switches`

```
---
# Group configuration for 'switches' group.
ansible_user: alice
```

Common Configuration Variables

The type of switch should be configured via the `switch_type` variable. See *Device-specific Configuration Variables* for details of the value to set for each device type.

`ansible_host` should be set to the management IP address used to access the device. `ansible_user` should be set to the user used to access the device.

Global switch configuration is specified via the `switch_config` variable. It should be a list of configuration lines to apply.

Per-interface configuration is specified via the `switch_interface_config` variable. It should be an object mapping switch interface names to configuration objects. Each configuration object contains a `description` item and a `config` item. The `config` item should contain a list of per-interface configuration lines.

The `switch_interface_config_enable_discovery` and `switch_interface_config_disable_discovery` variables take the same format as the `switch_interface_config` variable. They define interface configuration to apply to enable or disable hardware discovery of bare metal compute nodes.

Listing 22: `inventory/host_vars/switch0`

```
---
ansible_host: 1.2.3.4

ansible_user: alice

switch_config:
  - global config line 1
  - global config line 2

switch_interface_config:
  interface-0:
    description: controller0
    config:
      - interface-0 config line 1
      - interface-0 config line 2
  interface-1:
    description: compute0
    config:
      - interface-1 config line 1
      - interface-1 config line 2
```

Network device configuration can become quite repetitive, so it can be helpful to define group variables that can be referenced by multiple devices. For example:

Listing 23: inventory/group_vars/switches

```
---
# Group configuration for the 'switches' group.
switch_config_default:
  - default global config line 1
  - default global config line 2

switch_interface_config_controller:
  - controller interface config line 1
  - controller interface config line 2

switch_interface_config_compute:
  - compute interface config line 1
  - compute interface config line 2
```

Listing 24: inventory/host_vars/switch0

```
---
ansible_host: 1.2.3.4

ansible_user: alice

switch_config: "{{ switch_config_default }}"

switch_interface_config:
  interface-0:
    description: controller0
    config: "{{ switch_interface_config_controller }}"
  interface-1:
    description: compute0
    config: "{{ switch_interface_config_compute }}"
```

Device-specific Configuration Variables

Cumulus Linux (with NCLU)

Configuration for these devices is applied using the `nclu` Ansible module.

`switch_type` should be set to `nclu`.

SSH configuration

As with any non-switch host in the inventory, the `nclu` module relies on the default connection parameters used by Ansible:

- `ansible_host` is the hostname or IP address. Optional.
- `ansible_user` is the SSH username.

Dell OS6 and OS9

Configuration for these devices is applied using the `dellos6_config` and `dellos9_config` Ansible modules.

`switch_type` should be set to `dellos6` or `dellos9`.

Provider

- `ansible_host` is the hostname or IP address. Optional.
- `ansible_user` is the SSH username.
- `ansible_ssh_pass` is the SSH password.
- `switch_auth_pass` is the enable password.

Alternatively, set `switch_dellos_provider` to the value to be passed as the `provider` argument to the `dellos*_config` module.

Dell PowerConnect

Configuration for these devices is applied using the `stackhpc.dell-powerconnect-switch` Ansible role. The role uses the `expect` Ansible module to automate interaction with the switch CLI via SSH.

`switch_type` should be set to `dell-powerconnect`.

Provider

- `ansible_host` is the hostname or IP address. Optional.
- `ansible_user` is the SSH username.
- `switch_auth_pass` is the SSH password.

Juniper Junos OS

Configuration for these devices is applied using the `junos_config` Ansible module.

`switch_type` should be set to `junos`.

`switch_junos_config_format` may be used to set the format of the configuration. The variable is passed as the `src_format` argument to the `junos_config` module. The default value is `text`.

Provider

- `ansible_host` is the hostname or IP address. Optional.
- `ansible_user` is the SSH username.
- `ansible_ssh_pass` is the SSH password. Mutually exclusive with `ansible_ssh_private_key_file`.
- `ansible_ssh_private_key_file` is the SSH private key file. Mutually exclusive with `ansible_ssh_pass`.
- `switch_junos_timeout` may be set to a timeout in seconds for communicating with the device.

Alternatively, set `switch_junos_provider` to the value to be passed as the `provider` argument to the `junos_config` module.

Mellanox MLNX OS

Configuration for these devices is applied using the `stackhpc.mellanox-switch` Ansible role. The role uses the `expect` Ansible module to automate interaction with the switch CLI via SSH.

`switch_type` should be set to `mellanox`.

Provider

- `ansible_host` is the hostname or IP address. Optional.
- `ansible_user` is the SSH username.
- `switch_auth_pass` is the SSH password.

Network Configuration

Kayobe provides a flexible mechanism for configuring the networks in a system. Kayobe networks are assigned a name which is used as a prefix for variables that define the networks attributes. For example, to configure the `cidr` attribute of a network named `arpanet`, we would use a variable named `arpanet_cidr`.

Global Network Configuration

Global network configuration is stored in `${KAYOBE_CONFIG_PATH}/networks.yml`. The following attributes are supported:

`cidr` CIDR representation (<IP>/<prefix length>) of the networks IP subnet.

`allocation_pool_start` IP address of the start of Kayobes allocation pool range.

`allocation_pool_end` IP address of the end of Kayobes allocation pool range.

`inspection_allocation_pool_start` IP address of the start of ironic inspectors allocation pool range.

inspection_allocation_pool_end IP address of the end of ironic inspectors allocation pool range.

neutron_allocation_pool_start IP address of the start of neutrons allocation pool range.

neutron_allocation_pool_end IP address of the end of neutrons allocation pool range.

gateway IP address of the networks default gateway.

inspection_gateway IP address of the gateway for the hardware introspection network.

neutron_gateway IP address of the gateway for a neutron subnet based on this network.

vlan VLAN ID.

mtu Maximum Transmission Unit (MTU).

vip_address Virtual IP address (VIP) used by API services on this network.

fqdn Fully Qualified Domain Name (FQDN) used by API services on this network.

routes List of static IP routes. Each item should be a dict containing the item `cidr`, and optionally `gateway`, `table` and `options`. `cidr` is the CIDR representation of the routes destination. `gateway` is the IP address of the next hop. `table` is the name or ID of a routing table to which the route will be added. `options` is a list of option strings to add to the route.

rules List of IP routing rules. Each item should be an `iproute2` IP routing rule.

physical_network Name of the physical network on which this network exists. This aligns with the physical network concept in neutron.

libvirt_network_name A name to give to a Libvirt network representing this network on the seed hypervisor.

Configuring an IP Subnet

An IP subnet may be configured by setting the `cidr` attribute for a network to the CIDR representation of the subnet.

To configure a network called `example` with the `10.0.0.0/24` IP subnet:

Listing 25: `networks.yml`

```
example_cidr: 10.0.0.0/24
```

Configuring an IP Gateway

An IP gateway may be configured by setting the `gateway` attribute for a network to the IP address of the gateway.

To configure a network called `example` with a gateway at `10.0.0.1`:

Listing 26: `networks.yml`

```
example_gateway: 10.0.0.1
```

This gateway will be configured on all hosts to which the network is mapped. Note that configuring multiple IP gateways on a single host will lead to unpredictable results.

Configuring an API Virtual IP Address

A virtual IP (VIP) address may be configured for use by Kolla Ansible on the internal and external networks, on which the API services will be exposed. The variable will be passed through to the `kolla_internal_vip_address` or `kolla_external_vip_address` Kolla Ansible variable.

To configure a network called `example` with a VIP at `10.0.0.2`:

Listing 27: `networks.yml`

```
example_vip_address: 10.0.0.2
```

Configuring an API Fully Qualified Domain Name

A Fully Qualified Domain Name (FQDN) may be configured for use by Kolla Ansible on the internal and external networks, on which the API services will be exposed. The variable will be passed through to the `kolla_internal_fqdn` or `kolla_external_fqdn` Kolla Ansible variable.

To configure a network called `example` with an FQDN at `api.example.com`:

Listing 28: `networks.yml`

```
example_fqdn: api.example.com
```

Configuring Static IP Routes

Static IP routes may be configured by setting the `routes` attribute for a network to a list of routes.

To configure a network called `example` with a single IP route to the `10.1.0.0/24` subnet via `10.0.0.1`:

Listing 29: `networks.yml`

```
example_routes:
  - cidr: 10.1.0.0/24
    gateway: 10.0.0.1
```

These routes will be configured on all hosts to which the network is mapped.

If necessary, custom options may be added to the route:

Listing 30: `networks.yml`

```
example_routes:
  - cidr: 10.1.0.0/24
    gateway: 10.0.0.1
    options:
      - onlink
      - metric 400
```

Configuring a VLAN

A VLAN network may be configured by setting the `vlan` attribute for a network to the ID of the VLAN.

To configure a network called `example` with VLAN ID `123`:

Listing 31: `networks.yml`

```
example_vlan: 123
```

IP Address Allocation

IP addresses are allocated automatically by Kayobe from the allocation pool defined by `allocation_pool_start` and `allocation_pool_end`. If these variables are undefined, the entire network is used, except for network and broadcast addresses. IP addresses are only allocated if the network `cidr` is set and DHCP is not used (see `bootproto` in *Per-host Network Configuration*). The allocated addresses are stored in `${KAYOBE_CONFIG_PATH}/network-allocation.yml` using the global per-network attribute `ips` which maps Ansible inventory hostnames to allocated IPs.

If static IP address allocation is required, the IP allocation file `network-allocation.yml` may be manually populated with the required addresses.

Configuring Dynamic IP Address Allocation

To configure a network called `example` with the `10.0.0.0/24` IP subnet and an allocation pool spanning from `10.0.0.4` to `10.0.0.254`:

Listing 32: `networks.yml`

```
example_cidr: 10.0.0.0/24
example_allocation_pool_start: 10.0.0.4
example_allocation_pool_end: 10.0.0.254
```

Note: This pool should not overlap with an inspection or neutron allocation pool on the same network.

Configuring Static IP Address Allocation

To configure a network called `example` with statically allocated IP addresses for hosts `host1` and `host2`:

Listing 33: `network-allocation.yml`

```
example_ips:
  host1: 10.0.0.1
  host2: 10.0.0.2
```

Advanced: Policy-Based Routing

Policy-based routing can be useful in complex networking environments, particularly where asymmetric routes exist, and strict reverse path filtering is enabled.

Configuring IP Routing Tables

Custom IP routing tables may be configured by setting the global variable `network_route_tables` in `${KAYOBE_CONFIG_PATH}/networks.yml` to a list of route tables. These route tables will be added to `/etc/iproute2/rt_tables`.

To configure a routing table called `exampleroutetable` with ID `1`:

Listing 34: `networks.yml`

```
network_route_tables:
  - name: exampleroutetable
    id: 1
```

To configure route tables on specific hosts, use a host or group variables file.

Configuring IP Routing Policy Rules

IP routing policy rules may be configured by setting the `rules` attribute for a network to a list of rules. The format of a rule is the string which would be appended to `ip rule <add|del>` to create or delete the rule.

To configure a network called `example` with an IP routing policy rule to handle traffic from the subnet `10.1.0.0/24` using the routing table `exampleroutetable`:

Listing 35: `networks.yml`

```
example_rules:
  - from 10.1.0.0/24 table exampleroutetable
```

These rules will be configured on all hosts to which the network is mapped.

Configuring IP Routes on Specific Tables

A route may be added to a specific routing table by adding the name or ID of the table to a `table` attribute of the route:

To configure a network called `example` with a default route and a connected (local subnet) route to the subnet `10.1.0.0/24` on the table `exampleroutetable`:

Listing 36: `networks.yml`

```
example_routes:
  - cidr: 0.0.0.0/0
    gateway: 10.1.0.1
    table: exampleroutetable
```

(continues on next page)

(continued from previous page)

```
  - cidr: 10.1.0.0/24
    table: exampleroutetable
```

Per-host Network Configuration

Some network attributes are specific to a hosts role in the system, and these are stored in `${KAYOBE_CONFIG_PATH}/inventory/group_vars/<group>/network-interfaces`. The following attributes are supported:

`interface` The name of the network interface attached to the network.

`bootproto` Boot protocol for the interface. Valid values are `static` and `dhcp`. The default is `static`. When set to `dhcp`, an external DHCP server must be provided.

`defroute` Whether to set the interface as the default route. This attribute can be used to disable configuration of the default gateway by a specific interface. This is particularly useful to ignore a gateway address provided via DHCP. Should be set to a boolean value. The default is unset. This attribute is only supported on distributions of the Red Hat family.

`bridge_ports` For bridge interfaces, a list of names of network interfaces to add to the bridge.

`bond_mode` For bond interfaces, the bonds mode, e.g. 802.3ad.

`bond_slaves` For bond interfaces, a list of names of network interfaces to act as slaves for the bond.

`bond_miimon` For bond interfaces, the time in milliseconds between MII link monitoring.

`bond_updelay` For bond interfaces, the time in milliseconds to wait before declaring an interface up (should be multiple of `bond_miimon`).

`bond_downdelay` For bond interfaces, the time in milliseconds to wait before declaring an interface down (should be multiple of `bond_miimon`).

`bond_xmit_hash_policy` For bond interfaces, the xmit_hash_policy to use for the bond.

`bond_lacp_rate` For bond interfaces, the lacp_rate to use for the bond.

`ethtool_opts` Physical network interface options to apply with `ethtool`. When used on bond and bridge interfaces, settings apply to underlying interfaces. This should be a string of arguments passed to the `ethtool` utility, for example `"-G ${DEVICE} rx 8192 tx 8192"`.

IP Addresses

An interface will be assigned an IP address if the associated network has a `cidr` attribute. The IP address will be assigned from the range defined by the `allocation_pool_start` and `allocation_pool_end` attributes, if one has not been statically assigned in `network-allocation.yml`.

Configuring Ethernet Interfaces

An Ethernet interface may be configured by setting the `interface` attribute for a network to the name of the Ethernet interface.

To configure a network called `example` with an Ethernet interface on `eth0`:

Listing 37: `inventory/group_vars/<group>/network-interfaces`

```
example_interface: eth0
```

Configuring Bridge Interfaces

A Linux bridge interface may be configured by setting the `interface` attribute of a network to the name of the bridge interface, and the `bridge_ports` attribute to a list of interfaces which will be added as member ports on the bridge.

To configure a network called `example` with a bridge interface on `breth1`, and a single port `eth1`:

Listing 38: `inventory/group_vars/<group>/network-interfaces`

```
example_interface: breth1
example_bridge_ports:
  - eth1
```

Bridge member ports may be Ethernet interfaces, bond interfaces, or VLAN interfaces. In the case of bond interfaces, the bond must be configured separately in addition to the bridge, as a different named network. In the case of VLAN interfaces, the underlying Ethernet interface must be configured separately in addition to the bridge, as a different named network.

Configuring Bond Interfaces

A bonded interface may be configured by setting the `interface` attribute of a network to the name of the bonds master interface, and the `bond_slaves` attribute to a list of interfaces which will be added as slaves to the master.

To configure a network called `example` with a bond with master interface `bond0` and two slaves `eth0` and `eth1`:

Listing 39: `inventory/group_vars/<group>/network-interfaces`

```
example_interface: bond0
example_bond_slaves:
  - eth0
  - eth1
```

Optionally, the bond mode and MII monitoring interval may also be configured:

Listing 40: inventory/group_vars/<group>/network-interfaces

```
example_bond_mode: 802.3ad
example_bond_miimon: 100
```

Bond slaves may be Ethernet interfaces, or VLAN interfaces. In the case of VLAN interfaces, underlying Ethernet interface must be configured separately in addition to the bond, as a different named network.

Configuring VLAN Interfaces

A VLAN interface may be configured by setting the `interface` attribute of a network to the name of the VLAN interface. The interface name must be of the form `<parent interface>.<VLAN ID>`.

To configure a network called `example` with a VLAN interface with a parent interface of `eth2` for VLAN `123`:

Listing 41: inventory/group_vars/<group>/network-interfaces

```
example_interface: eth2.123
```

To keep the configuration DRY, reference the networks `vlan` attribute:

Listing 42: inventory/group_vars/<group>/network-interfaces

```
example_interface: "eth2.{{ example_vlan }}"
```

Ethernet interfaces, bridges, and bond master interfaces may all be parents to a VLAN interface.

Bridges and VLANs

Adding a VLAN interface to a bridge directly will allow tagged traffic for that VLAN to be forwarded by the bridge, whereas adding a VLAN interface to an Ethernet or bond interface that is a bridge member port will prevent tagged traffic for that VLAN being forwarded by the bridge.

For example, if you are bridging `eth1` to `breth1` and want to access VLAN 1234 as a tagged VLAN from the host, while still allowing Neutron to access traffic for that VLAN via Open vSwitch, your setup should look like this:

```
$ sudo brctl show
bridge name     bridge id               STP enabled     interfaces
breth1          8000.56e6b95b4178       no              p-breth1-phy
                                                        eth1

$ sudo ip addr show | grep 1234 | head -1
10: breth1.1234@breth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc␣
↪noqueue state UP group default qlen 1000
```

It should **not** look like this:

```
$ sudo brctl show
bridge name     bridge id               STP enabled     interfaces
breth1          8000.56e6b95b4178       no              p-breth1-phy
                                                        eth1
$ sudo ip addr show | grep 1234 | head -1
10: eth1.1234@eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc␣
↪noqueue state UP group default qlen 1000
```

This second configuration may be desirable to prevent specific traffic, e.g. of the internal API network, from reaching Neutron.

Domain Name Service (DNS) Resolver Configuration

Kayobe supports configuration of hosts DNS resolver via `resolv.conf`. DNS configuration should be added to `dns.yml`. For example:

Listing 43: `dns.yml`

```
resolv_nameservers:
  - 8.8.8.8
  - 8.8.4.4
resolv_domain: example.com
resolv_search:
  - kayobe.example.com
```

It is also possible to prevent kayobe from modifying `resolv.conf` by setting `resolv_is_managed` to `false`.

Network Role Configuration

In order to provide flexibility in the systems network topology, Kayobe maps the named networks to logical network roles. A single named network may perform multiple roles, or even none at all. The available roles are:

Overcloud admin network (`admin_oc_net_name`) Name of the network used to access the overcloud for admin purposes, e.g for remote SSH access.

Overcloud out-of-band network (`oob_oc_net_name`) Name of the network used by the seed to access the out-of-band management controllers of the bare metal overcloud hosts.

Overcloud provisioning network (`provision_oc_net_name`) Name of the network used by the seed to provision the bare metal overcloud hosts.

Workload out-of-band network (`oob_wl_net_name`) Name of the network used by the overcloud hosts to access the out-of-band management controllers of the bare metal workload hosts.

Workload provisioning network (`provision_wl_net_name`) Name of the network used by the overcloud hosts to provision the bare metal workload hosts.

Workload cleaning network (`cleaning_net_name`) Name of the network used by the overcloud hosts to clean the baremetal workload hosts.

Internal network (`internal_net_name`) Name of the network used to expose the internal OpenStack API endpoints.

Public network (`public_net_name`) Name of the network used to expose the public OpenStack API endpoints.

Tunnel network (`tunnel_net_name`) Name of the network used by Neutron to carry tenant overlay network traffic.

External networks (`external_net_names`, deprecated: `external_net_name`) List of names of networks used to provide external network access via Neutron. If `external_net_name` is defined, `external_net_names` defaults to a list containing only that network.

Storage network (`storage_net_name`) Name of the network used to carry storage data traffic.

Storage management network (`storage_mgmt_net_name`) Name of the network used to carry storage management traffic.

Swift storage network (`swift_storage_net_name`) Name of the network used to carry Swift storage data traffic. Defaults to the storage network (`storage_net_name`).

Swift storage replication network (`swift_storage_replication_net_name`) Name of the network used to carry storage management traffic. Defaults to the storage management network (`storage_mgmt_net_name`)

Workload inspection network (`inspection_net_name`) Name of the network used to perform hardware introspection on the bare metal workload hosts.

These roles are configured in `${KAYOBE_CONFIG_PATH}/networks.yml`.

Configuring Network Roles

To configure network roles in a system with two networks, `example1` and `example2`:

Listing 44: `networks.yml`

```
admin_oc_net_name: example1
oob_oc_net_name: example1
provision_oc_net_name: example1
oob_wl_net_name: example1
provision_wl_net_name: example2
internal_net_name: example2
public_net_name: example2
tunnel_net_name: example2
external_net_names:
  - example2
storage_net_name: example2
storage_mgmt_net_name: example2
swift_storage_net_name: example2
swift_replication_net_name: example2
inspection_net_name: example2
cleaning_net_name: example2
```

Overcloud Admin Network

The admin network is intended to be used for remote access to the overcloud hosts. Kayobe will use the address assigned to the host on this network as the `ansible_host` when executing playbooks. It is therefore a necessary requirement to configure this network.

By default Kayobe will use the overcloud provisioning network as the admin network. It is, however, possible to configure a separate network. To do so, you should override `admin_oc_net_name` in your networking configuration.

If a separate network is configured, the following requirements should be taken into consideration:

- The admin network must be configured to use the same physical network interface as the provisioning network. This is because the PXE MAC address is used to lookup the interface for the cloud-init network configuration that occurs during bifrost provisioning of the overcloud.

Overcloud Provisioning Network

If using a seed to inspect the bare metal overcloud hosts, it is necessary to define a DHCP allocation pool for the seeds ironic inspector DHCP server using the `inspection_allocation_pool_start` and `inspection_allocation_pool_end` attributes of the overcloud provisioning network.

Note: This example assumes that the `example` network is mapped to `provision_oc_net_name`.

To configure a network called `example` with an inspection allocation pool:

```
example_inspection_allocation_pool_start: 10.0.0.128
example_inspection_allocation_pool_end: 10.0.0.254
```

Note: This pool should not overlap with a kayobe allocation pool on the same network.

Workload Cleaning Network

A separate cleaning network, which is used by the overcloud to clean baremetal workload (compute) hosts, may optionally be specified. Otherwise, the Workload Provisoning network is used. It is necessary to define an IP allocation pool for neutron using the `neutron_allocation_pool_start` and `neutron_allocation_pool_end` attributes of the cleaning network. This controls the IP addresses that are assigned to workload hosts during cleaning.

Note: This example assumes that the `example` network is mapped to `cleaning_net_name`.

To configure a network called `example` with a neutron provisioning allocation pool:

```
example_neutron_allocation_pool_start: 10.0.1.128
example_neutron_allocation_pool_end: 10.0.1.195
```

Note: This pool should not overlap with a kayobe or inspection allocation pool on the same network.

Workload Provisioning Network

If using the overcloud to provision bare metal workload (compute) hosts, it is necessary to define an IP allocation pool for the overclouds neutron provisioning network using the `neutron_allocation_pool_start` and `neutron_allocation_pool_end` attributes of the workload provisioning network.

Note: This example assumes that the `example` network is mapped to `provision_wl_net_name`.

To configure a network called `example` with a neutron provisioning allocation pool:

```
example_neutron_allocation_pool_start: 10.0.1.128
example_neutron_allocation_pool_end: 10.0.1.195
```

Note: This pool should not overlap with a kayobe or inspection allocation pool on the same network.

Workload Inspection Network

If using the overcloud to inspect bare metal workload (compute) hosts, it is necessary to define a DHCP allocation pool for the overclouds ironic inspector DHCP server using the `inspection_allocation_pool_start` and `inspection_allocation_pool_end` attributes of the workload provisioning network.

Note: This example assumes that the `example` network is mapped to `provision_wl_net_name`.

To configure a network called `example` with an inspection allocation pool:

```
example_inspection_allocation_pool_start: 10.0.1.196
example_inspection_allocation_pool_end: 10.0.1.254
```

Note: This pool should not overlap with a kayobe or neutron allocation pool on the same network.

Neutron Networking

Note: This assumes the use of the neutron `openvswitch` ML2 mechanism driver for control plane networking.

Certain modes of operation of neutron require layer 2 access to physical networks in the system. Hosts in the `network` group (by default, this is the same as the `controllers` group) run the neutron networking services (Open vSwitch agent, DHCP agent, L3 agent, metadata agent, etc.).

The kayobe network configuration must ensure that the neutron Open vSwitch bridges on the network hosts have access to the external network. If bare metal compute nodes are in use, then they must also have access to the workload provisioning network. This can be done by ensuring that the external and workload provisioning network interfaces are bridges. Kayobe will ensure connectivity between these Linux bridges and the neutron Open vSwitch bridges via a virtual Ethernet pair. See *Configuring Bridge Interfaces*.

Network to Host Mapping

Networks are mapped to hosts using the variable `network_interfaces`. Kayobes playbook group variables define some sensible defaults for this variable for hosts in the top level standard groups. These defaults are set using the network roles typically required by the group.

Seed

By default, the seed is attached to the following networks:

- overcloud admin network
- overcloud out-of-band network
- overcloud provisioning network

This list may be extended by setting `seed_extra_network_interfaces` to a list of names of additional networks to attach. Alternatively, the list may be completely overridden by setting `seed_network_interfaces`. These variables are found in `${KAYOBE_CONFIG_PATH}/seed.yml`.

Seed Hypervisor

By default, the seed hypervisor is attached to the same networks as the seed.

This list may be extended by setting `seed_hypervisor_extra_network_interfaces` to a list of names of additional networks to attach. Alternatively, the list may be completely overridden by setting `seed_hypervisor_network_interfaces`. These variables are found in `${KAYOBE_CONFIG_PATH}/seed-hypervisor.yml`.

Controllers

By default, controllers are attached to the following networks:

- overcloud admin network
- workload (compute) out-of-band network
- workload (compute) provisioning network
- workload (compute) inspection network
- workload (compute) cleaning network
- internal network
- storage network

In addition, if the controllers are also in the `network` group, they are attached to the following networks:

- public network
- external network
- tunnel network

This list may be extended by setting `controller_extra_network_interfaces` to a list of names of additional networks to attach. Alternatively, the list may be completely overridden by setting `controller_network_interfaces`. These variables are found in `${KAYOBE_CONFIG_PATH}/controllers.yml`.

Network Hosts

By default, controllers provide Neutron network services and load balancing. If separate network hosts are used (see *Example 1: Adding Network Hosts*), they are attached to the following networks:

- overcloud admin network
- internal network
- storage network
- public network
- external network
- tunnel network

This list may be extended by setting `controller_network_host_extra_network_interfaces` to a list of names of additional networks to attach. Alternatively, the list may be completely overridden by setting `controller_network_host_network_interfaces`. These variables are found in `${KAYOBE_CONFIG_PATH}/controllers.yml`.

Monitoring Hosts

By default, the monitoring hosts are attached to the same networks as the controllers when they are in the `controllers` group. If the monitoring hosts are not in the `controllers` group, they are attached to the following networks by default:

- overcloud admin network
- internal network
- public network

This list may be extended by setting `monitoring_extra_network_interfaces` to a list of names of additional networks to attach. Alternatively, the list may be completely overridden by setting `monitoring_network_interfaces`. These variables are found in `${KAYOBE_CONFIG_PATH}/monitoring.yml`.

Storage Hosts

By default, the storage hosts are attached to the following networks:

- overcloud admin network
- internal network
- storage network
- storage management network

In addition, if Swift is enabled, they can also be attached to the Swift management and replication networks.

Virtualised Compute Hosts

By default, virtualised compute hosts are attached to the following networks:

- overcloud admin network
- internal network
- storage network
- tunnel network

This list may be extended by setting `compute_extra_network_interfaces` to a list of names of additional networks to attach. Alternatively, the list may be completely overridden by setting `compute_network_interfaces`. These variables are found in `${KAYOBE_CONFIG_PATH}/compute.yml`.

Other Hosts

If additional hosts are managed by kayobe, the networks to which these hosts are attached may be defined in a host or group variables file. See *Control Plane Service Placement* for further details.

Complete Example

The following example combines the complete network configuration into a single system configuration. In our example cloud we have three networks: `management`, `cloud` and `external`:

The `management` network is used to access the servers BMCs and by the seed to inspect and provision the cloud hosts. The `cloud` network carries all internal control plane and storage traffic, and is used by the control plane to provision the bare metal compute hosts. Finally, the `external` network links the cloud to the outside world.

We could describe such a network as follows:

Listing 45: `networks.yml`

```
---
# Network role mappings.
oob_oc_net_name: management
provision_oc_net_name: management
oob_wl_net_name: management
provision_wl_net_name: cloud
internal_net_name: cloud
public_net_name: external
external_net_name: external
storage_net_name: cloud
storage_mgmt_net_name: cloud
inspection_net_name: cloud

# management network definition.
management_cidr: 10.0.0.0/24
management_allocation_pool_start: 10.0.0.1
management_allocation_pool_end: 10.0.0.127
management_inspection_allocation_pool_start: 10.0.0.128
management_inspection_allocation_pool_end: 10.0.0.254

# cloud network definition.
cloud_cidr: 10.0.1.0/24
cloud_allocation_pool_start: 10.0.1.1
cloud_allocation_pool_end: 10.0.1.127
cloud_inspection_allocation_pool_start: 10.0.1.128
cloud_inspection_allocation_pool_end: 10.0.1.195
cloud_neutron_allocation_pool_start: 10.0.1.196
cloud_neutron_allocation_pool_end: 10.0.1.254

# external network definition.
external_cidr: 10.0.3.0/24
external_allocation_pool_start: 10.0.3.1
external_allocation_pool_end: 10.0.3.127
external_neutron_allocation_pool_start: 10.0.3.128
external_neutron_allocation_pool_end: 10.0.3.254
external_routes:
  - cidr: 10.0.4.0/24
    gateway: 10.0.3.1
```

We can map these networks to network interfaces on the seed and controller hosts:

Listing 46: `inventory/group_vars/seed/network-interfaces`

```
---
management_interface: eth0
```

Listing 47: `inventory/group_vars/controllers/network-interfaces`

```
---
management_interface: eth0
cloud_interface: breth1
```

(continues on next page)

(continued from previous page)

```
cloud_bridge_ports:
  - eth1
external_interface: eth2
```

We have defined a bridge for the cloud network on the controllers as this will allow it to be plugged into a neutron Open vSwitch bridge.

Kayobe will allocate IP addresses for the hosts that it manages:

Listing 48: `network-allocation.yml`

```
---
management_ips:
  seed: 10.0.0.1
  control0: 10.0.0.2
  control1: 10.0.0.3
  control2: 10.0.0.4
cloud_ips:
  control0: 10.0.1.1
  control1: 10.0.1.2
  control2: 10.0.1.3
external_ips:
  control0: 10.0.3.1
  control1: 10.0.3.2
  control2: 10.0.3.3
```

Note that although this file does not need to be created manually, doing so allows for a predictable IP address mapping which may be desirable in some cases.

Host Configuration

This section covers configuration of hosts. It does not cover configuration or deployment of containers. Hosts that are configured by Kayobe include:

- Seed hypervisor (`kayobe seed hypervisor host configure`)
- Seed (`kayobe seed host configure`)
- Overcloud (`kayobe overcloud host configure`)

Unless otherwise stated, all host configuration described here is applied to each of these types of host.

See also:

Ansible tags for limiting the scope of Kayobe commands are included under the relevant sections of this page (for more information see *Tags*).

Configuration Location

Some host configuration options are set via global variables, and others have a variable for each type of host. The latter variables are included in the following files under `${KAYOBE_CONFIG_PATH}`:

- `seed-hypervisor.yml`
- `seed.yml`
- `compute.yml`
- `controller.yml`
- `monitoring.yml`
- `storage.yml`

Note that any variable may be set on a per-host or per-group basis, by using inventory host or group variables - these delineations are for convenience.

Paths

Several directories are used by Kayobe on the remote hosts. There is a hierarchy of variables in `${KAYOBE_CONFIG_PATH}/globals.yml` that can be used to control where these are located.

- `base_path` (default `/opt/kayobe/`) sets the default base path for various directories.
- `config_path` (default `{{ base_path }}/etc`) is a path in which to store configuration files.
- `image_cache_path` (default `{{ base_path }}/images`) is a path in which to cache downloaded or built images.
- `source_checkout_path` (default `{{ base_path }}/src`) is a path into which to store clones of source code repositories.
- `virtualenv_path` (default `{{ base_path }}/venvs`) is a path in which to create Python virtual environments.

SSH Known Hosts

tags:

`ssh-known-host`

While strictly this configuration is applied to the Ansible control host (`localhost`), it is applied during the `host configure` commands. The `ansible_host` of each host is added as an SSH known host. This is typically the hosts IP address on the admin network (`admin_oc_net_name`), as defined in `${KAYOBE_CONFIG_PATH}/network-allocation.yml` (see *IP Address Allocation*).

Kayobe User Bootstrapping

tags:

`kayobe-ansible-user`

Kayobe uses a user account defined by the `kayobe_ansible_user` variable (in `${KAYOBE_CONFIG_PATH}/globals.yml`) for remote SSH access. By default, this is `stack`.

Typically, the image used to provision these hosts will not include this user account, so Kayobe performs a bootstrapping step to create it, as a different user. In cloud images, there is often a user named after the OS distro, e.g. `centos` or `ubuntu`. This user defaults to the name of the user running Kayobe, but may be set via the following variables:

- `seed_hypervisor_bootstrap_user`
- `seed_bootstrap_user`
- `compute_bootstrap_user`
- `controller_bootstrap_user`
- `monitoring_bootstrap_user`
- `storage_bootstrap_user`

For example, to set the bootstrap user for controllers to `centos`:

Listing 49: `controllers.yml`

```
controller_bootstrap_user: centos
```

PyPI Mirror and proxy

tags:

`pip`

Kayobe supports configuration of a PyPI mirror and/or proxy, via variables in `${KAYOBE_CONFIG_PATH}/pip.yml`. Mirror functionality is enabled by setting the `pip_local_mirror` variable to `true` and proxy functionality is enabled by setting `pip_proxy` variable to a proxy URL.

Kayobe will generate configuration for:

- `pip` to use the mirror and proxy
- `easy_install` to use the mirror

for the list of users defined by `pip_applicable_users` (default `kayobe_ansible_user` and `root`), in addition to the user used for Kolla Ansible (`kolla_ansible_user`). The mirror URL is configured via `pip_index_url`, and `pip_trusted_hosts` is a list of trusted hosts, for which SSL verification will be disabled.

For example, to configure use of the test PyPI mirror at https://test.pypi.org/simple/:

Listing 50: `pip.yml`

```
pip_local_mirror: true
pip_index_url: https://test.pypi.org/simple/
```

To configure use of the PyPI proxy:

Listing 51: `pip.yml`

```
pip_proxy: http://your_proxy_server:3128
```

Kayobe Remote Virtual Environment

tags:

`kayobe-target-venv`

By default, Ansible executes modules remotely using the system python interpreter, even if the Ansible control process is executed from within a virtual environment (unless the `local` connection plugin is used). This is not ideal if there are python dependencies that must be installed with isolation from the system python packages. Ansible can be configured to use a virtualenv by setting the host variable `ansible_python_interpreter` to a path to a python interpreter in an existing virtual environment.

If kayobe detects that `ansible_python_interpreter` is set and references a virtual environment, it will create the virtual environment if it does not exist. Typically this variable should be set via a group variable in the inventory for hosts in the `seed`, `seed-hypervisor`, and/or `overcloud` groups.

The default Kayobe configuration in the `kayobe-config` repository sets `ansible_python_interpreter` to `{{ virtualenv_path }}/kayobe/bin/python` for the `seed`, `seed-hypervisor`, and `overcloud` groups.

Disk Wiping

tags:

`wipe-disks`

Using hosts that may have stale data on their disks could affect the deployment of the cloud. This is not a configuration option, since it should only be performed once to avoid losing useful data. It is triggered by passing the `--wipe-disks` argument to the `host configure` commands.

Users and Groups

tags:

`users`

Linux user accounts and groups can be configured using the `users_default` variable in `${KAYOBE_CONFIG_PATH}/users.yml`. The format of the list is that used by the `users` variable of the singleplatform-eng.users role. The following variables can be used to set the users for specific types of hosts:

- seed_hypervisor_users
- seed_users
- compute_users
- controller_users
- monitoring_users
- storage_users

In the following example, a single user named bob is created. A password hash has been generated via mkpasswd --method=sha-512. The user is added to the wheel group, and an SSH key is authorised. The SSH public key should be added to the Kayobe configuration.

Listing 52: users.yml

```
users_default:
  - username: bob
    name: Bob
    password: "$6$wJt9MLWrHlWN8
↪$oXJHbdaslm9guD5EC3Dry1mphuqF9NPeQ43OXk3cXZa2ze/
↪F9FOTxm2KvvDkbdxBDs7ouwdiLTUJ1Ff40.cFU."
    groups:
      - wheel
    append: True
    ssh_key:
      - "{{ lookup('file', kayobe_config_path ~ '/ssh-keys/id_rsa_bob.pub')␣
↪}}"
```

Package Repositories

tags:

dnf

Kayobe supports configuration of package repositories via DNF, via variables in ${KAYOBE_CONFIG_PATH}/dnf.yml.

Configuration of dnf.conf

Global configuration of DNF is stored in /etc/dnf/dnf.conf, and options can be set via the dnf_config variable. Options are added to the [main] section of the file. For example, to configure DNF to use a proxy server:

Listing 53: dnf.yml

```
dnf_config:
  proxy: https://proxy.example.com
```

CentOS and EPEL Mirrors

CentOS and EPEL mirrors can be enabled by setting dnf_use_local_mirror to true. CentOS repository mirrors are configured via the following variables:

- dnf_centos_mirror_host (default mirror.centos.org) is the mirror hostname.
- dnf_centos_mirror_directory (default centos) is a directory on the mirror in which repositories may be accessed.

EPEL repository mirrors are configured via the following variables:

- dnf_epel_mirror_host (default download.fedoraproject.org) is the mirror hostname.
- dnf_epel_mirror_directory (default pub/epel) is a directory on the mirror in which repositories may be accessed.

For example, to configure CentOS and EPEL mirrors at mirror.example.com:

Listing 54: dnf.yml

```
dnf_use_local_mirror: true
dnf_centos_mirror_host: mirror.example.com
dnf_epel_mirror_host: mirror.example.com
```

Custom DNF Repositories

It is also possible to configure a list of custom DNF repositories via the dnf_custom_repos variable. The format is a dict/map, with repository names mapping to a dict/map of arguments to pass to the Ansible yum_repository module.

For example, the following configuration defines a single DNF repository called widgets.

Listing 55: `dnf.yml`

```
dnf_custom_repos:
  widgets:
    baseurl: http://example.com/repo
    file: widgets
    gpgkey: http://example.com/gpgkey
    gpgcheck: yes
```

Disabling EPEL

It is possible to disable the EPEL DNF repository by setting `dnf_install_epel` to `false`.

DNF Automatic

DNF Automatic provides a mechanism for applying regular updates of packages. DNF Automatic is disabled by default, and may be enabled by setting `dnf_automatic_enabled` to `true`.

Listing 56: `dnf.yml`

```
dnf_automatic_enabled: true
```

By default, only security updates are applied. Updates for all packages may be installed by setting `dnf_automatic_upgrade_type` to `default`. This may cause the system to be less predictable as packages are updated without oversight or testing.

SELinux

tags:

`disable-selinux`

SELinux is not supported by Kolla Ansible currently, so it is disabled by Kayobe. If necessary, Kayobe will reboot systems in order to apply a change to the SELinux configuration. The timeout for waiting for systems to reboot is `disable_selinux_reboot_timeout`. Alternatively, the reboot may be avoided by setting `disable_selinux_do_reboot` to `false`.

Network Configuration

tags:

`network`

Configuration of host networking is covered in depth in *Network Configuration*.

Sysctls

tags:

`sysctl`

Arbitrary `sysctl` configuration can be applied to hosts. The variable format is a dict/map, mapping parameter names to their required values. The following variables can be used to set `sysctl` configuration specific types of hosts:

- `seed_hypervisor_sysctl_parameters`
- `seed_sysctl_parameters`
- `compute_sysctl_parameters`
- `controller_sysctl_parameters`
- `monitoring_sysctl_parameters`
- `storage_sysctl_parameters`

For example, to set the `net.ipv4.ip_forward` parameter to `1` on controllers:

Listing 57: `controllers.yml`

```
controller_sysctl_parameters:
  net.ipv4.ip_forward: 1
```

Disable cloud-init

tags:

`disable-cloud-init`

cloud-init is a popular service for performing system bootstrapping. If you are not using cloud-init, this section can be skipped.

If using the seeds Bifrost service to provision the control plane hosts, the use of cloud-init may be configured via the `kolla_bifrost_dib_init_element` variable.

cloud-init searches for network configuration in order of increasing precedence; each item overriding the previous. In some cases, on subsequent boots cloud-init can automatically reconfigure network interfaces and cause some issues in network configuration. To disable cloud-init from running after the initial server bootstrapping, set `disable_cloud_init` to `true` in `${KAYOBE_CONFIG_PATH}/overcloud.yml`.

Disable Glean

tags:

`disable-glean`

The `glean` service can be used to perform system bootstrapping, serving a similar role to `cloud-init`. If you are not using `glean`, this section can be skipped.

If using the seeds Bifrost service to provision the control plane hosts, the use of `glean` may be configured via the `kolla_bifrost_dib_init_element` variable.

After the initial server bootstrapping, the glean service can cause problems as it attempts to enable all network interfaces, which can lead to timeouts while booting. To avoid this, the `glean` service is disabled. Additionally, any network interface configuration files generated by `glean` and not overwritten by Kayobe are removed.

Timezone

tags:

`timezone`

The timezone can be configured via the `timezone` variable in `${KAYOBE_CONFIG_PATH}/time.yml`. The value must be a valid Linux timezone. For example:

Listing 58: `time.yml`

```
timezone: Europe/London
```

NTP

Since the Ussuri release, Kayobe no longer supports configuration of an NTP daemon on the host, since the `ntp` package is no longer available in CentOS 8.

Kolla Ansible can deploy a chrony container on overcloud hosts, and from the Ussuri release chrony is enabled by default. There is no support for running a chrony container on the seed or seed hypervisor hosts.

To disable the containerised chrony daemon, set the following in `${KAYOBE_CONFIG_PATH}/kolla.yml`:

```
kolla_enable_chrony: false
```

Software RAID

tags:

`mdadm`

While it is possible to use RAID directly with LVM, some operators may prefer the userspace tools provided by `mdadm` or may have existing software RAID arrays they want to manage with Kayobe.

Software RAID arrays may be configured via the `mdadm_arrays` variable. For convenience, this is mapped to the following variables:

- `seed_hypervisor_mdadm_arrays`
- `seed_mdadm_arrays`
- `compute_mdadm_arrays`
- `controller_mdadm_arrays`
- `monitoring_mdadm_arrays`
- `storage_mdadm_arrays`

The format of these variables is as defined by the `mdadm_arrays` variable of the mrlesmithjr.mdadm Ansible role.

For example, to configure two of the seeds disks as a RAID1 `mdadm` array available as `/dev/md0`:

Listing 59: `seed.yml`

```
seed_mdadm_arrays:
  - name: md0
    devices:
      - /dev/sdb
      - /dev/sdc
    level: '1'
    state: present
```

Encryption

tags:

`luks`

Encrypted block devices may be configured via the `luks_devices` variable. For convenience, this is mapped to the following variables:

- `seed_hypervisor_luks_devices`
- `seed_luks_devices`
- `compute_luks_devices`
- `controller_luks_devices`
- `monitoring_luks_devices`
- `storage_luks_devices`

The format of these variables is as defined by the `luks_devices` variable of the stackhpc.luks Ansible role.

For example, to encrypt the software raid device, `/dev/md0`, on the seed, and make it available as `/dev/mapper/md0crypt`

Listing 60: `seed.yml`

```
seed_luks_devices:
  - name: md0crypt
    device: /dev/md0
```

Note: It is not yet possible to encrypt the root device.

LVM

tags:

`lvm`

Logical Volume Manager (LVM) physical volumes, volume groups, and logical volumes may be configured via the `lvm_groups` variable. For convenience, this is mapped to the following variables:

- `seed_hypervisor_lvm_groups`
- `seed_lvm_groups`
- `compute_lvm_groups`
- `controller_lvm_groups`
- `monitoring_lvm_groups`
- `storage_lvm_groups`

The format of these variables is as defined by the `lvm_groups` variable of the mrlesmithjr.manage-lvm Ansible role.

LVM for libvirt

LVM is not configured by default on the seed hypervisor. It is possible to configure LVM to provide storage for a `libvirt` storage pool, typically mounted at `/var/lib/libvirt/images`.

To use this configuration, set the `seed_hypervisor_lvm_groups` variable to `"{{ seed_hypervisor_lvm_groups_with_data }}"` and provide a list of disks via the `seed_hypervisor_lvm_group_data_disks` variable.

LVM for Docker

Note: In Train and earlier releases of Kayobe, the `data` volume group was always enabled by default.

If the `devicemapper` Docker storage driver is in use, the default LVM configuration is optimised for it. The `devicemapper` driver requires a thin provisioned LVM volume. A second logical volume is used for storing Docker volume data, mounted at `/var/lib/docker/volumes`. Both logical volumes are created from a single `data` volume group.

This configuration is enabled by the following variables, which default to `true` if the `devicemapper` driver is in use or `false` otherwise:

- `compute_lvm_group_data_enabled`
- `controller_lvm_group_data_enabled`
- `seed_lvm_group_data_enabled`
- `storage_lvm_group_data_enabled`

These variables can be set to `true` to enable the data volume group if the `devicemapper` driver is not in use. This may be useful where the `docker-volumes` logical volume is required.

To use this configuration, a list of disks must be configured via the following variables:

- `seed_lvm_group_data_disks`
- `compute_lvm_group_data_disks`
- `controller_lvm_group_data_disks`
- `monitoring_lvm_group_data_disks`
- `storage_lvm_group_data_disks`

For example, to configure two of the seeds disks for use by LVM:

Listing 61: `seed.yml`

```
seed_lvm_group_data_disks:
  - /dev/sdb
  - /dev/sdc
```

The Docker volumes LVM volume is assigned a size given by the following variables, with a default value of 75% (of the volume groups capacity):

- `seed_lvm_group_data_lv_docker_volumes_size`
- `compute_lvm_group_data_lv_docker_volumes_size`
- `controller_lvm_group_data_lv_docker_volumes_size`
- `monitoring_lvm_group_data_lv_docker_volumes_size`
- `storage_lvm_group_data_lv_docker_volumes_size`

If using a Docker storage driver other than `devicemapper`, the remaining 25% of the volume group can be used for Docker volume data. In this case, the LVM volumes size can be increased to 100%:

Listing 62: controllers.yml

```
controller_lvm_group_data_lv_docker_volumes_size: 100%
```

If using a Docker storage driver other than devicemapper, it is possible to avoid using LVM entirely, thus avoiding the requirement for multiple disks. In this case, set the appropriate <host>_lvm_groups variable to an empty list:

Listing 63: storage.yml

```
storage_lvm_groups: []
```

Custom LVM

To define additional logical logical volumes in the default data volume group, modify one of the following variables:

- seed_lvm_group_data_lvs
- compute_lvm_group_data_lvs
- controller_lvm_group_data_lvs
- monitoring_lvm_group_data_lvs
- storage_lvm_group_data_lvs

Include the variable <host>_lvm_group_data_lv_docker_volumes in the list to include the LVM volume for Docker volume data:

Listing 64: monitoring.yml

```
monitoring_lvm_group_data_lvs:
  - "{{ monitoring_lvm_group_data_lv_docker_volumes }}"
  - lvname: other-vol
    size: 1%
    create: true
    filesystem: ext4
    mount: true
    mntp: /path/to/mount
```

It is possible to define additional LVM volume groups via the following variables:

- seed_lvm_groups_extra
- compute_lvm_groups_extra
- controller_lvm_groups_extra
- monitoring_lvm_groups_extra
- storage_lvm_groups_extra

For example:

Listing 65: `compute.yml`

```
compute_lvm_groups_extra:
  - vgname: other-vg
    disks:
      - /dev/sdb
    create: true
    lvnames:
      - lvname: other-vol
        size: 100%FREE
        create: true
        mount: false
```

Alternatively, replace the entire volume group list via one of the `<host>_lvm_groups` variables to replace the default configuration with a custom one.

Listing 66: `controllers.yml`

```
controller_lvm_groups:
  - vgname: only-vg
    disks: /dev/sdb
    create: true
    lvnames:
      - lvname: only-vol
        size: 100%
        create: true
        mount: false
```

Kolla-Ansible bootstrap-servers

Kolla Ansible provides some host configuration functionality via the `bootstrap-servers` command, which may be leveraged by Kayobe.

See the Kolla Ansible documentation for more information on the functions performed by this command, and how to configure it.

Note that from the Ussuri release, Kayobe creates a user account for Kolla Ansible rather than this being done by Kolla Ansible during `bootstrap-servers`. See *User account creation* for details.

Kolla-Ansible Remote Virtual Environment

tags:

`kolla-ansible`
`kolla-target-venv`

See *Context: Remote Execution Environment* for information about remote Python virtual environments for Kolla Ansible.

Docker Engine

tags:

`docker`

Docker engine configuration is applied by both Kayobe and Kolla Ansible (during bootstrap-servers).

The `docker_storage_driver` variable sets the Docker storage driver, and by default the `overlay2` driver is used. If using the `devicemapper` driver, see *LVM* for information about configuring LVM for Docker.

Various options are defined in `${KAYOBE_CONFIG_PATH}/docker.yml` for configuring the `devicemapper` storage.

A private Docker registry may be configured via `docker_registry`, with a Certificate Authority (CA) file configured via `docker_registry_ca`.

To use one or more Docker Registry mirrors, use the `docker_registry_mirrors` variable.

If using an MTU other than 1500, `docker_daemon_mtu` can be used to configure this. This setting does not apply to containers using `net=host` (as Kolla Ansibles containers do), but may be necessary when building images.

Dockers live restore feature can be configured via `docker_daemon_live_restore`, although it is disabled by default due to issues observed.

Kolla Configuration

Anyone using Kayobe to build images should familiarise themselves with the Kolla projects documentation.

Container Image Build Host

Images are built on hosts in the `container-image-builders` group. The default Kayobe Ansible inventory places the seed host in this group, although it is possible to put a different host in the group, by modifying the inventory.

For example, to build images on `localhost`:

Listing 67: `inventory/groups`

```
[container-image-builders:children]
```

Listing 68: `inventory/hosts`

```
[container-image-builders]
localhost
```

Kolla Installation

Prior to building container images, Kolla and its dependencies will be installed on the container image build host. The following variables affect the installation of Kolla:

kolla_ctl_install_type Type of installation, either `binary` (PyPI) or `source` (git). Default is `source`.

kolla_source_path Path to directory for Kolla source code checkout. Default is `{{ source_checkout_path ~ '/kolla' }}`.

kolla_source_url URL of Kolla source code repository if type is `source`. Default is https://opendev.org/openstack/kolla.

kolla_source_version Version (branch, tag, etc.) of Kolla source code repository if type is `source`. Default is `{{ openstack_branch }}`, which is the same as the Kayobe upstream branch name.

kolla_venv Path to virtualenv in which to install Kolla on the container image build host. Default is `{{ virtualenv_path ~ '/kolla' }}`.

kolla_build_config_path Path in which to generate kolla configuration. Default is `{{ config_path ~ '/kolla' }}`.

For example, to install from a custom Git repository:

Listing 69: `kolla.yml`

```
kolla_source_url: https://git.example.com/kolla
kolla_source_version: downstream
```

Global Configuration

The following variables are global, affecting all container images. They are used to generate the Kolla configuration file, `kolla-build.conf`, and also affect *Kolla Ansible configuration*.

kolla_base_distro Kolla base container image distribution. Default is `centos`.

kolla_install_type Kolla container image type: `binary` or `source`. Default is `binary`.

kolla_docker_namespace Docker namespace to use for Kolla images. Default is `kolla`.

kolla_docker_registry URL of docker registry to use for Kolla images. Default is to use the value of `docker_registry` variable (see *Docker Engine*).

kolla_docker_registry_username Username to use to access a docker registry. Default is not set, in which case the registry will be used without authentication.

kolla_docker_registry_password Password to use to access a docker registry. Default is not set, in which case the registry will be used without authentication.

kolla_openstack_release Kolla OpenStack release version. This should be a Docker image tag. Default is the OpenStack release name (e.g. `rocky`) on stable branches and tagged releases, or `master` on the Kayobe `master` branch.

kolla_tag Kolla container image tag. This is the tag that will be applied to built container images. Default is `kolla_openstack_release`.

For example, to build the Kolla `centos` `binary` images with a namespace of `example`, and a private Docker registry at `registry.example.com:4000`, tagged with `7.0.0.1`:

Listing 70: `kolla.yml`

```
kolla_base_distro: centos
kolla_install_type: binary
kolla_docker_namespace: example
kolla_docker_registry: registry.example.com:4000
kolla_openstack_release: 7.0.0.1
```

The `ironic-api` image built with this configuration would be referenced as follows:

```
registry.example.com:4000/example/centos-binary-ironic-api:7.0.0.1
```

Further customisation of the Kolla configuration file can be performed by writing a file at `${KAYOBE_CONFIG_PATH/kolla/kolla-build.conf`. For example, to enable debug logging:

Listing 71: `kolla/kolla-build.conf`

```
[DEFAULT]
debug = True
```

Seed Images

The `kayobe seed container image build` command builds images for the seed services. The only image required for the seed services is the `bifrost-deploy` image.

Overcloud Images

The `kayobe overcloud container image build` command builds images for the control plane. The default set of images built depends on which services and features are enabled via the `kolla_enable_<service>` flags in `$KAYOBE_CONFIG_PATH/kolla.yml`.

For example, the following configuration will enable the Magnum service and add the `magnum-api` and `magnum-conductor` containers to the set of overcloud images that will be built:

Listing 72: `kolla.yml`

```
kolla_enable_magnum: true
```

If a required image is not built when the corresponding flag is set, check the image sets defined in `overcloud_container_image_sets` in `ansible/group_vars/all/kolla`.

Image Customisation

There are three main approaches to customising the Kolla container images:

1. Overriding Jinja2 blocks
2. Overriding Jinja2 variables
3. Source code locations

Overriding Jinja2 blocks

Kollas images are defined via Jinja2 templates that generate Dockerfiles. Jinja2 blocks are frequently used to allow specific statements in one or more Dockerfiles to be replaced with custom statements. See the Kolla documentation for details.

Blocks are configured via the `kolla_build_blocks` variable, which is a dict mapping Jinja2 block names in to their contents.

For example, to override the block `header` to add a custom label to every image:

Listing 73: `kolla.yml`

```
kolla_build_blocks:
  header: |
    LABEL foo="bar"
```

This will result in Kayobe generating a `template-override.j2` file with the following content:

Listing 74: `template-override.j2`

```
{% extends parent_template %}

{% block header %}
LABEL foo="bar"
{% endblock %}
```

Overriding Jinja2 variables

Jinja2 variables offer another way to customise images. See the Kolla documentation for details of using variable overrides to modify the list of packages to install in an image.

Variable overrides are configured via the `kolla_build_customizations` variable, which is a dict/map mapping names of variables to override to their values.

For example, to add `mod_auth_openidc` to the list of packages installed in the `keystone-base` image, we can set the variable `keystone_base_packages_append` to a list containing `mod_auth_openidc`.

Listing 75: `kolla.yml`

```
kolla_build_customizations:
  keystone_base_packages_append:
    - mod_auth_openidc
```

This will result in Kayobe generating a `template-override.j2` file with the following content:

Listing 76: `template-override.j2`

```
{% extends parent_template %}

{% set keystone_base_packages_append = ["mod_auth_openidc"] %}
```

Note that the variable value will be JSON-encoded in `template-override.j2`.

Source code locations

For `source` image builds, configuration of source code locations for packages installed in containers by Kolla is possible via the `kolla_sources` variable. The format is a dict/map mapping names of sources to their definitions. See the Kolla documentation for details. The default is to specify the URL and version of Bifrost, as defined in `${KAYOBE_CONFIG_PATH}/bifrost.yml`.

For example, to specify a custom source location for the `ironic-base` package:

Listing 77: `kolla.yml`

```
kolla_sources:
  bifrost-base:
    type: "git"
    location: "{{ kolla_bifrost_source_url }}"
    reference: "{{ kolla_bifrost_source_version }}"
  ironic-base:
    type: "git"
    location: https://git.example.com/ironic
    reference: downstream
```

This will result in Kayobe adding the following configuration to `kolla-build.conf`:

Listing 78: `kolla-build.conf`

```
[bifrost-base]
type = git
location = https://opendev.org/openstack/bifrost
reference = stable/rocky

[ironic-base]
type = git
location = https://git.example.com/ironic
reference = downstream
```

Note that it is currently necessary to include the Bifrost source location if using a seed.

Plugins & additions

These features can also be used for installing plugins and additions to `source` type images.

For example, to install a `networking-ansible` plugin in the `neutron-server` image:

Listing 79: `kolla.yml`

```
kolla_sources:
  bifrost-base:
    type: "git"
    location: "{{ kolla_bifrost_source_url }}"
    reference: "{{ kolla_bifrost_source_version }}"
  neutron-server-plugin-networking-ansible:
    type: "git"
    location: https://git.example.com/networking-ansible
    reference: downstream
```

The `neutron-server` image automatically installs any plugins provided to it. For images that do not, a block such as the following may be required:

Listing 80: kolla.yml

```
kolla_build_blocks:
  neutron_server_footer: |
    ADD plugins-archive /
    pip --no-cache-dir install /plugins/*
```

A similar approach may be used for additions.

Kolla Ansible Configuration

Kayobe relies heavily on Kolla Ansible for deployment of the OpenStack control plane. Kolla Ansible is installed locally on the Ansible control host (the host from which Kayobe commands are executed), and Kolla Ansible commands are executed from there.

Kolla Ansible configuration is stored in `${KAYOBE_CONFIG_PATH}/kolla.yml`.

Configuration of Ansible

Ansible configuration is described in detail in the Ansible documentation. In addition to the standard locations, Kayobe supports using an Ansible configuration file located in the Kayobe configuration at `${KAYOBE_CONFIG_PATH}/kolla/ansible.cfg` or `${KAYOBE_CONFIG_PATH}/ansible.cfg`. Note that if the `ANSIBLE_CONFIG` environment variable is specified it takes precedence over this file.

Kolla Ansible Installation

Prior to deploying containers, Kolla Ansible and its dependencies will be installed on the Ansible control host. The following variables affect the installation of Kolla Ansible:

kolla_ansible_ctl_install_type Type of Kolla Ansible control installation. One of `binary` (PyPI) or `source` (git). Default is `source`.

kolla_ansible_source_url URL of Kolla Ansible source code repository if type is `source`. Default is https://opendev.org/openstack/kolla-ansible.

kolla_ansible_source_version Version (branch, tag, etc.) of Kolla Ansible source code repository if type is `source`. Default is the same as the Kayobe upstream branch.

kolla_ansible_venv_extra_requirements Extra requirements to install inside the Kolla Ansible virtualenv. Default is an empty list.

kolla_upper_constraints_file Upper constraints file for installation of Kolla. Default is `{{ pip_upper_constraints_file }}`, which has a default of `https://releases.openstack.org/constraints/upper/{{ openstack_branch }}`.

Example: custom git repository

To install Kolla Ansible from a custom git repository:

Listing 81: `$KAYOBE_CONFIG_PATH/kolla.yml`

```
kolla_ansible_source_url: https://git.example.com/kolla-ansible
kolla_ansible_source_version: downstream
```

Virtual Environment Extra Requirements

Extra Python packages can be installed inside the Kolla Ansible virtualenv, such as when required by Ansible plugins.

For example, to use the hashi_vault Ansible lookup plugin, its `hvac` dependency can be installed using:

Listing 82: `$KAYOBE_CONFIG_PATH/kolla.yml`

```
---
# Extra requirements to install inside the Kolla Ansible virtualenv.
kolla_ansible_venv_extra_requirements:
  - "hvac"
```

Local environment

The following variables affect the local environment on the Ansible control host. They reference environment variables, and should be configured using those rather than modifying the Ansible variable directly. The file `kayobe-env` in the kayobe-config git repository sets some sensible defaults for these variables, based on the recommended environment directory structure.

`kolla_ansible_source_path` Path to directory for Kolla Ansible source code checkout. Default is `$KOLLA_SOURCE_PATH`, or `$PWD/src/kolla-ansible`.

`kolla_ansible_venv` Path to virtualenv in which to install Kolla Ansible on the Ansible control host. Default is `$KOLLA_VENV_PATH` or `$PWD/venvs/kolla-ansible`.

`kolla_config_path` Path to Kolla Ansible configuration directory. Default is `$KOLLA_CONFIG_PATH` or `/etc/kolla`.

Global Configuration

The following variables are global, affecting all containers. They are used to generate the Kolla Ansible configuration file, `globals.yml`, and also affect *Kolla image build configuration*.

Kolla Images

The following variables affect which Kolla images are used, and how they are accessed.

kolla_base_distro Kolla base container image distribution. Default is `centos`.

kolla_install_type Kolla container image type: `binary` or `source`. Default is `binary`.

kolla_docker_registry URL of docker registry to use for Kolla images. Default is not set, in which case Dockerhub will be used.

kolla_docker_namespace Docker namespace to use for Kolla images. Default is `kolla`.

kolla_docker_registry_username Username to use to access a docker registry. Default is not set, in which case the registry will be used without authentication.

kolla_docker_registry_password Password to use to access a docker registry. Default is not set, in which case the registry will be used without authentication.

kolla_openstack_release Kolla OpenStack release version. This should be a Docker image tag. Default is `{{ openstack_release }}`, which takes the OpenStack release name (e.g. `rocky`) on stable branches and tagged releases, or `master` on the Kayobe `master` branch.

For example, to deploy Kolla `centos binary` images with a namespace of `example`, and a private Docker registry at `registry.example.com:4000`, tagged with `7.0.0.1`:

Listing 83: `$KAYOBE_CONFIG_PATH/kolla.yml`

```
kolla_base_distro: centos
kolla_install_type: binary
kolla_docker_namespace: example
kolla_docker_registry: registry.example.com:4000
kolla_openstack_release: 7.0.0.1
```

The deployed `ironic-api` image would be referenced as follows:

```
registry.example.com:4000/example/centos-binary-ironic-api:7.0.0.1
```

Ansible

The following variables affect how Ansible accesses the remote hosts.

kolla_ansible_user User account to use for Kolla SSH access. Default is `kolla`.

kolla_ansible_group Primary group of Kolla SSH user. Default is `kolla`.

kolla_ansible_become Whether to use privilege escalation for all operations performed via Kolla Ansible. Default is `false` since the 8.0.0 Ussuri release.

kolla_ansible_target_venv Path to a virtual environment on remote hosts to use for Ansible module execution. Default is `{{ virtualenv_path }}/kolla-ansible`. May be set to `None` to use the system Python interpreter.

Context: Remote Execution Environment

By default, Ansible executes modules remotely using the system python interpreter, even if the Ansible control process is executed from within a virtual environment (unless the `local` connection plugin is used). This is not ideal if there are python dependencies that must be installed with isolation from the system python packages. Ansible can be configured to use a virtualenv by setting the host variable `ansible_python_interpreter` to a path to a python interpreter in an existing virtual environment.

The variable `kolla_ansible_target_venv` configures the use of a virtual environment on the remote hosts. The default configuration should work in most cases.

User account creation

Since the Ussuri release, Kayobe creates a user account for Kolla Ansible rather than this being done during Kolla Ansibles `bootstrap-servers` command. This workflow is more compatible with Ansible fact caching, but does mean that Kolla Ansibles `create_kolla_user` variable cannot be used to disable creation of the user account. Instead, set `kolla_ansible_create_user` to `false`.

`kolla_ansible_create_user` Whether to create a user account, configure passwordless sudo and authorise an SSH key for Kolla Ansible. Default is `true`.

OpenStack Logging

The following variable affects OpenStack debug logging.

`kolla_openstack_logging_debug` Whether debug logging is enabled for OpenStack services. Default is `false`.

Example: enabling debug logging

In certain situations it may be necessary to enable debug logging for all OpenStack services. This is not usually advisable in production.

Listing 84: `$KAYOBE_CONFIG_PATH/kolla.yml`

```
---
kolla_openstack_logging_debug: true
```

TLS Encryption of APIs

The following variables affect TLS encryption of the public API.

`kolla_enable_tls_external` Whether TLS is enabled for the public API endpoints. Default is `no`.

`kolla_external_tls_cert` A TLS certificate bundle to use for the public API endpoints, if `kolla_enable_tls_external` is `true`. Note that this should be formatted as a literal style block scalar.

kolla_external_fqdn_cacert Path to a CA certificate file to use for the `OS_CACERT` environment variable in openrc files when TLS is enabled, instead of Kolla Ansibles default.

The following variables affect TLS encryption of the internal API. Currently this requires all Kolla images to be built with the APIs root CA trusted.

kolla_enable_tls_internal Whether TLS is enabled for the internal API endpoints. Default is `no`.

kolla_internal_tls_cert A TLS certificate bundle to use for the internal API endpoints, if `kolla_enable_tls_internal` is `true`. Note that this should be formatted as a literal style block scalar.

kolla_internal_fqdn_cacert Path to a CA certificate file to use for the `OS_CACERT` environment variable in openrc files when TLS is enabled, instead of Kolla Ansibles default.

Example: enabling TLS for the public API

It is highly recommended to use TLS encryption to secure the public API. Here is an example:

Listing 85: `$KAYOBE_CONFIG_PATH/kolla.yml`

```
---
kolla_enable_tls_external: yes
kolla_external_tls_cert: |
  -----BEGIN CERTIFICATE-----
  ...
  -----END CERTIFICATE-----
kolla_external_fqdn_cacert: /path/to/ca/certificate/bundle
```

Example: enabling TLS for the internal API

It is highly recommended to use TLS encryption to secure the internal API. Here is an example:

Listing 86: `$KAYOBE_CONFIG_PATH/kolla.yml`

```
---
kolla_enable_tls_internal: yes
kolla_internal_tls_cert: |
  -----BEGIN CERTIFICATE-----
  ...
  -----END CERTIFICATE-----
kolla_internal_fqdn_cacert: /path/to/ca/certificate/bundle
```

Other certificates

In general, Kolla Ansible expects certificates to be in a directory configured via `kolla_certificates_dir`, which defaults to a directory named `certificates` in the same directory as `globals.yml`. Kayobe follows this pattern, and will pass files and directories added to `${KAYOBE_CONFIG_PATH}/kolla/certificates/` through to Kolla Ansible. This can be useful when enabling backend API TLS encryption, or providing custom CA certificates to be added to the trust store in containers. It is also possible to use this path to provide certificate bundles for the external or internal APIs, as an alternative to `kolla_external_tls_cert` and `kolla_internal_tls_cert`.

Note that Ansible will automatically decrypt these files if they are encrypted via Ansible Vault and it has access to a Vault password.

Example: adding a trusted custom CA certificate to containers

In an environment with a private CA, it may be necessary to add the root CA certificate to the trust store of containers.

Listing 87: `$KAYOBE_CONFIG_PATH`

```
kolla/
  certificates/
    ca/
      private-ca.crt
```

These files should be PEM-formatted, and have a `.crt` extension.

Example: adding certificates for backend TLS

Kolla Ansible backend TLS can be used to provide end-to-end encryption of API traffic.

Listing 88: `$KAYOBE_CONFIG_PATH`

```
kolla/
  certificates/
    backend-cert.pem
    backend-key.pem
```

See the Kolla Ansible documentation for how to provide service and/or host-specific certificates and keys.

Custom Global Variables

Kolla Ansible uses a single file for global variables, `globals.yml`. Kayobe provides configuration variables for all required variables and many of the most commonly used the variables in this file. Some of these are in `$KAYOBE_CONFIG_PATH/kolla.yml`, and others are determined from other sources such as the networking configuration in `$KAYOBE_CONFIG_PATH/networks.yml`.

Additional global configuration may be provided by creating `$KAYOBE_CONFIG_PATH/kolla/globals.yml`. Variables in this file will be templated using Jinja2, and merged with the Kayobe `globals.yml` configuration.

Example: use a specific tag for each image

For more fine-grained control over images, Kolla Ansible allows a tag to be defined for each image. For example, for `nova-api`:

Listing 89: `$KAYOBE_CONFIG_PATH/kolla/globals.yml`

```
---
# Use a custom tag for the nova-api container image.
nova_api_tag: v1.2.3
```

Example: debug logging per-service

Enabling debug logging globally can lead to a lot of additional logs being generated. Often we are only interested in a particular service. For example, to enable debug logging for Nova services:

Listing 90: `$KAYOBE_CONFIG_PATH/kolla/globals.yml`

```
---
nova_logging_debug: true
```

Host variables

Kayobe generates a host_vars file for each host in the Kolla Ansible inventory. These contain network interfaces and other host-specific things.

`kolla_seed_inventory_pass_through_host_vars` List of names of host variables to pass through from kayobe hosts to the Kolla Ansible seed host, if set. See also `kolla_seed_inventory_pass_through_host_vars_map`. The default is:

```
kolla_seed_inventory_pass_through_host_vars:
  - "ansible_host"
  - "ansible_port"
  - "ansible_ssh_private_key_file"
  - "kolla_api_interface"
  - "kolla_bifrost_network_interface"
```

kolla_seed_inventory_pass_through_host_vars_map Dict mapping names of variables in `kolla_seed_inventory_pass_through_host_vars` to the variable to use in Kolla Ansible. If a variable name is not in this mapping the kayobe name is used. The default is:

```
kolla_seed_inventory_pass_through_host_vars_map:
  kolla_api_interface: "api_interface"
  kolla_bifrost_network_interface: "bifrost_network_interface"
```

kolla_overcloud_inventory_pass_through_host_vars List of names of host variables to pass through from Kayobe hosts to Kolla Ansible hosts, if set. See also `kolla_overcloud_inventory_pass_through_host_vars_map`. The default is:

```
kolla_overcloud_inventory_pass_through_host_vars:
  - "ansible_host"
  - "ansible_port"
  - "ansible_ssh_private_key_file"
  - "kolla_network_interface"
  - "kolla_api_interface"
  - "kolla_storage_interface"
  - "kolla_cluster_interface"
  - "kolla_swift_storage_interface"
  - "kolla_swift_replication_interface"
  - "kolla_provision_interface"
  - "kolla_inspector_dnsmasq_interface"
  - "kolla_dns_interface"
  - "kolla_tunnel_interface"
  - "kolla_external_vip_interface"
  - "kolla_neutron_external_interfaces"
  - "kolla_neutron_bridge_names"
```

kolla_overcloud_inventory_pass_through_host_vars_map Dict mapping names of variables in `kolla_overcloud_inventory_pass_through_host_vars` to the variable to use in Kolla Ansible. If a variable name is not in this mapping the Kayobe name is used. The default is:

```
kolla_overcloud_inventory_pass_through_host_vars_map:
  kolla_network_interface: "network_interface"
  kolla_api_interface: "api_interface"
  kolla_storage_interface: "storage_interface"
  kolla_cluster_interface: "cluster_interface"
  kolla_swift_storage_interface: "swift_storage_interface"
  kolla_swift_replication_interface: "swift_replication_interface"
  kolla_provision_interface: "provision_interface"
  kolla_inspector_dnsmasq_interface: "ironic_dnsmasq_interface"
  kolla_dns_interface: "dns_interface"
  kolla_tunnel_interface: "tunnel_interface"
  kolla_neutron_external_interfaces: "neutron_external_interface"
  kolla_neutron_bridge_names: "neutron_bridge_name"
```

Custom Group Variables

Group variables can be used to set configuration for all hosts in a group. They can be set in Kolla Ansible by placing files in `${KAYOBE_CONFIG_PATH}/kolla/inventory/group_vars/*`. Since this directory is copied directly into the Kolla Ansible inventory, Kolla Ansible group names should be used. It should be noted that `extra-vars` and `host_vars` take precedence over `group_vars`. For more information on variable precedence see the Ansible documentation.

Example: configure a Nova cell

In Kolla Ansible, Nova cells are configured via group variables. For example, to configure `cell0001` the following file could be created:

Listing 91: `$KAYOBE_CONFIG_PATH/kolla/inventory/group_vars/cell0001/all`

```
---
nova_cell_name: cell0001
nova_cell_novncproxy_group: cell0001-vnc
nova_cell_conductor_group: cell0001-control
nova_cell_compute_group: cell0001-compute
```

Passwords

Kolla Ansible auto-generates passwords to a file, `passwords.yml`. Kayobe handles the orchestration of this, as well as encryption of the file using an Ansible Vault password specified in the `KAYOBE_VAULT_PASSWORD` environment variable, if present. The file is generated to `$KAYOBE_CONFIG_PATH/kolla/passwords.yml`, and should be stored along with other Kayobe configuration files. This file should not be manually modified.

kolla_ansible_custom_passwords Dictionary containing custom passwords to add or override in the Kolla passwords file. Default is `{{ kolla_ansible_default_custom_passwords }}`, which contains SSH keys for use by Kolla Ansible and Bifrost.

Configuring Custom Passwords

In order to write additional passwords to `passwords.yml`, set the kayobe variable `kolla_ansible_custom_passwords` in `$KAYOBE_CONFIG_PATH/kolla.yml`.

Listing 92: `$KAYOBE_CONFIG_PATH/kolla.yml`

```
---
# Dictionary containing custom passwords to add or override in the Kolla
# passwords file.
kolla_ansible_custom_passwords: >
  {{ kolla_ansible_default_custom_passwords |
     combine({'my_custom_password': 'correcthorsebatterystaple'}) }}
```

Control Plane Services

Kolla Ansible provides a flexible mechanism for configuring the services that it deploys. Kayobe adds some commonly required configuration options to the defaults provided by Kolla Ansible, but also allows for the free-form configuration supported by Kolla Ansible. The Kolla Ansible documentation should be used as a reference.

Enabling Services

Services deployed by Kolla Ansible are enabled via flags.

kolla_enable_<service or feature> There are various flags that can be used to enable features. These map to variables named `enable_<service or feature>` in Kolla Ansible. The default set of enabled services and features is the same as in Kolla ansible, except that Ironic is enabled by default in Kayobe.

Example: enabling a service

A common task is enabling a new OpenStack service. This may be done via the `kolla_enable_*` flags, for example:

Listing 93: `$KAYOBE_CONFIG_PATH/kolla.yml`

```
---
kolla_enable_swift: true
```

Note that in some cases additional configuration may be required to successfully deploy a service - check the Kolla Ansible configuration reference.

Service Configuration

Kolla-ansibles flexible configuration is described in the Kolla Ansible service configuration documentation. We wont duplicate that here, but essentially it involves creating files under a directory which for users of kayobe will be `$KOLLA_CONFIG_PATH/config`. In kayobe, files in this directory are auto-generated and managed by kayobe. Instead, users should create files under `$KAYOBE_CONFIG_PATH/kolla/config` with the same directory structure. These files will be templated using Jinja2, merged with kayobes own configuration, and written out to `$KOLLA_CONFIG_PATH/config`.

The following files, if present, will be templated and provided to Kolla Ansible. All paths are relative to `$KAYOBE_CONFIG_PATH/kolla/config`. Note that typically Kolla Ansible does not use the same wildcard patterns, and has a more restricted set of files that it will process. In some cases, it may be necessary to inspect the Kolla Ansible configuration tasks to determine which files are supported.

Table 1: Kolla-ansible configuration files

File	Purpose
`aodh.conf`	Aodh configuration.
`aodh/*`	Extended Aodh configuration.

continues on next page

Table 1 – continued from previous page

File	Purpose
backup.my.cnf	Mariabackup configuration.
barbican.conf	Barbican configuration.
barbican/*	Extended Barbican configuration.
blazar.conf	Blazar configuration.
blazar/*	Extended Blazar configuration.
ceilometer.conf	Ceilometer configuration.
ceilometer/*	Extended Ceilometer configuration.
cinder.conf	Cinder configuration.
cinder/*	Extended Cinder configuration.
cloudkitty.conf	CloudKitty configuration.
cloudkitty/*	Extended CloudKitty configuration.
designate.conf	Designate configuration.
designate/*	Extended Designate configuration.
elasticsearch/*	Elasticsearch configuration.
fluentd/filter	Fluentd filter configuration.
fluentd/input	Fluentd input configuration.
fluentd/output	Fluentd output configuration.
galera.cnf	MariaDB configuration.
glance.conf	Glance configuration.
glance/*	Extended Glance configuration.
global.conf	Global configuration for all OpenStack services.
gnocchi.conf	Gnocchi configuration.
gnocchi/*	Extended Gnocchi configuration.
grafana.ini	Grafana configuration.
grafana/*	Extended Grafana configuration.
haproxy/*	Main HAProxy configuration.
haproxy-config/*	Modular HAProxy configuration.
heat.conf	Heat configuration.
heat/*	Extended heat configuration.
horizon/*	Extended horizon configuration.
influx*	InfluxDB configuration.
ironic-inspector.conf	Ironic inspector configuration.
ironic.conf	Ironic configuration.
ironic/*	Extended ironic configuration.
kafka.server.properties	Kafka configuration.
kafka/*	Extended Kafka configuration.
keepalived/*	Extended keepalived configuration.
keystone.conf	Keystone configuration.
keystone/*	Extended keystone configuration.
magnum.conf	Magnum configuration.
magnum/*	Extended magnum configuration.
manila.conf	Manila configuration.
manila/*	Extended manila configuration.
mariadb/*	Extended MariaDB configuration.
masakari.conf	Masakari configuration.
masakari/*	Extended masakari configuration.
monasca/*	Extended Monasca configuration.

continues on next page

Table 1 – continued from previous page

File	Purpose
murano.conf	Murano configuration.
murano/*	Extended murano configuration.
neutron.conf	Neutron configuration.
neutron/ml2_conf.ini	Neutron ML2 configuration.
neutron/*	Extended neutron configuration.
nova.conf	Nova configuration.
nova/*	Extended nova configuration.
octavia.conf	Octavia configuration.
octavia/*	Extended Octavia configuration.
prometheus/*	Prometheus configuration.
sahara.conf	Sahara configuration.
sahara/*	Extended sahara configuration.
storm/*	Extended Storm configuration.
swift/*	Extended swift configuration.
zookeeper.cfg	Zookeeper configuration.
zookeeper/*	Extended Zookeeper configuration.

Configuring an OpenStack Component

To provide custom configuration to be applied to all glance services, create `$KAYOBE_CONFIG_PATH/kolla/config/glance.conf`. For example:

Listing 94: `$KAYOBE_CONFIG_PATH/kolla/config/glance.conf`

```
[DEFAULT]
api_limit_max = 500
```

Configuring an OpenStack Service

To provide custom configuration for the glance API service, create `$KAYOBE_CONFIG_PATH/kolla/config/glance/glance-api.conf`. For example:

Listing 95: `$KAYOBE_CONFIG_PATH/kolla/config/glance/glance-api.conf`

```
[DEFAULT]
api_limit_max = 500
```

Bifrost

This section covers configuration of the Bifrost service that runs on the seed host. Bifrost configuration is typically applied in `${KAYOBE_CONFIG_PATH}/bifrost.yml`. Consult the Bifrost documentation for further details of Bifrost usage and configuration.

Bifrost installation

Note: This section may be skipped if using an upstream Bifrost container image.

The following options are used if building the Bifrost container image locally.

`kolla_bifrost_source_url` URL of Bifrost source code repository. Default is https://opendev.org/openstack/bifrost.

`kolla_bifrost_source_version` Version (branch, tag, etc.) of Bifrost source code repository. Default is `{{ openstack_branch }}`, which is the same as the Kayobe upstream branch name.

For example, to install Bifrost from a custom git repository:

Listing 96: `bifrost.yml`

```
kolla_bifrost_source_url: https://git.example.com/bifrost
kolla_bifrost_source_version: downstream
```

Overcloud root disk image configuration

Bifrost uses Diskimage builder (DIB) to build a root disk image that is deployed to overcloud hosts when they are provisioned. The following options configure how this image is built. Consult the Diskimage-builder documentation for further information on building disk images.

The default configuration builds a CentOS 8 whole disk (partitioned) image with SELinux disabled and a serial console enabled. Cloud-init is used to process the configuration drive built by Bifrost, rather than the Bifrost default of simple-init.

`kolla_bifrost_dib_os_element` DIB base OS element. Default is `centos`.

`kolla_bifrost_dib_os_release` DIB image OS release. Default is `8-stream`.

`kolla_bifrost_dib_elements_default` *Added in the Train release. Use kolla_bifrost_dib_elements in earlier releases.*

List of default DIB elements. Default is `["disable-selinux", "enable-serial-console", "vm"]`. The `vm` element is poorly named, and causes DIB to build a whole disk image rather than a single partition.

kolla_bifrost_dib_elements_extra *Added in the Train release. Use kolla_bifrost_dib_elements in earlier releases.*

List of additional DIB elements. Default is none.

kolla_bifrost_dib_elements List of DIB elements. Default is a combination of `kolla_bifrost_dib_elements_default` and `kolla_bifrost_dib_elements_extra`.

kolla_bifrost_dib_init_element DIB init element. Default is `cloud-init-datasources`.

kolla_bifrost_dib_env_vars_default *Added in the Train release. Use kolla_bifrost_dib_env_vars in earlier releases.*

DIB default environment variables. Default is `{"DIB_CLOUD_INIT_DATASOURCES": "ConfigDrive"}`.

kolla_bifrost_dib_env_vars_extra *Added in the Train release. Use kolla_bifrost_dib_env_vars in earlier releases.*

DIB additional environment variables. Default is none.

kolla_bifrost_dib_env_vars DIB environment variables. Default is combination of `kolla_bifrost_dib_env_vars_default` and `kolla_bifrost_dib_env_vars_extra`.

kolla_bifrost_dib_packages List of DIB packages to install. Default is to install no extra packages.

The disk image is built during the deployment of seed services. It is worth noting that currently, the image will not be rebuilt if it already exists. To force rebuilding the image, it is necessary to remove the file. On the seed:

```
docker exec bifrost_deploy rm /httpboot/deployment_image.qcow2
```

Then on the control host:

```
(kayobe) $ kayobe seed service deploy
```

Example: Adding an element

In the following, we extend the list of DIB elements to add the `growpart` element:

Listing 97: `bifrost.yml`

```
kolla_bifrost_dib_elements_extra:
  - "growpart"
```

Example: Building an XFS root filesystem image

By default, DIB will format the image as `ext4`. In some cases it might be useful to use XFS, for example when using the `overlay` Docker storage driver which can reach the maximum number of hardlinks allowed by `ext4`.

In DIB, we achieve this by setting the `FS_TYPE` environment variable to `xfs`.

Listing 98: `bifrost.yml`

```
kolla_bifrost_dib_env_vars_extra:
  FS_TYPE: "xfs"
```

Example: Configuring a development user account

Warning: A development user account should not be used in production.

When debugging a failed deployment, it can sometimes be necessary to allow access to the image via a preconfigured user account with a known password. This can be achieved via the devuser element.

This example shows how to add the `devuser` element, and configure a username and password for an account that has passwordless sudo:

Listing 99: `bifrost.yml`

```
kolla_bifrost_dib_elements_extra:
  - "devuser"

kolla_bifrost_dib_env_vars_extra:
  DIB_DEV_USER_USERNAME: "devuser"
  DIB_DEV_USER_PASSWORD: "correct horse battery staple"
  DIB_DEV_USER_PWDLESS_SUDO: "yes"
```

Alternatively, the dynamic-login element can be used to authorize SSH keys by appending them to the kernel arguments.

Example: Installing a package

It can be necessary to install additional packages in the root disk image. Rather than needing to write a custom DIB element, we can use the `kolla_bifrost_dib_packages` variable. For example, to install the `biosdevname` package:

Listing 100: `bifrost.yml`

```
kolla_bifrost_dib_packages:
  - "biosdevname"
```

Ironic configuration

The following options configure the Ironic service in the `bifrost-deploy` container.

`kolla_bifrost_enabled_hardware_types` List of hardware types to enable for Bifrosts Ironic. Default is `["ipmi"]`.

`kolla_bifrost_extra_kernel_options` List of extra kernel parameters for Bifrosts Ironic PXE configuration. Default is none.

Ironic Inspector configuration

The following options configure the Ironic Inspector service in the `bifrost-deploy` container.

`kolla_bifrost_inspector_processing_hooks` List of of inspector processing plugins. Default is `{{ inspector_processing_hooks }}`, defined in `${KAYOBE_CONFIG_PATH}/inspector.yml`.

`kolla_bifrost_inspector_port_addition` Which MAC addresses to add as ports during introspection. One of `all`, `active` or `pxe`. Default is `{{ inspector_add_ports }}`, defined in `${KAYOBE_CONFIG_PATH}/inspector.yml`.

`kolla_bifrost_inspector_extra_kernel_options` List of extra kernel parameters for the inspector default PXE configuration. Default is `{{ inspector_extra_kernel_options }}`, defined in `${KAYOBE_CONFIG_PATH}/inspector.yml`. When customising this variable, the default extra kernel parameters should be kept to retain full node inspection capabilities.

kolla_bifrost_inspector_rules List of introspection rules for Bifrosts Ironic Inspector service. Default is `{{ inspector_rules }}`, defined in `${KAYOBE_CONFIG_PATH}/inspector.yml`.

kolla_bifrost_inspector_ipmi_username Ironic inspector IPMI username to set via an introspection rule. Default is `{{ ipmi_username }}`, defined in `${KAYOBE_CONFIG_PATH}/bmc.yml`.

kolla_bifrost_inspector_ipmi_password Ironic inspector IPMI password to set via an introspection rule. Default is `{{ ipmi_password }}`, defined in `${KAYOBE_CONFIG_PATH}/bmc.yml`.

kolla_bifrost_inspector_lldp_switch_port_interface Ironic inspector network interface name on which to check for an LLDP switch port description to use as the nodes name. Default is `{{ inspector_lldp_switch_port_interface_default }}`, defined in `${KAYOBE_CONFIG_PATH}/inspector.yml`.

kolla_bifrost_inspector_deploy_kernel Ironic inspector deployment kernel location. Default is `http://{{ provision_oc_net_name | net_ip }}:8080/ipa.kernel`.

kolla_bifrost_inspector_deploy_ramdisk Ironic inspector deployment ramdisk location. Default is `http://{{ provision_oc_net_name | net_ip }}:8080/ipa.initramfs`.

kolla_bifrost_inspection_timeout Timeout of hardware inspection on overcloud nodes, in seconds. Default is `{{ inspector_inspection_timeout }}`, defined in `${KAYOBE_CONFIG_PATH}/inspector.yml`.

Ironic Python Agent (IPA) configuration

Note: If building IPA images locally (`ipa_build_images` is `true`) this section can be skipped.

The following options configure the source of Ironic Python Agent images used by Bifrost for inspection and deployment. Consult the Ironic Python Agent documentation for full details.

kolla_bifrost_ipa_kernel_upstream_url URL of Ironic Python Agent (IPA) kernel image. Default is `{{ inspector_ipa_kernel_upstream_url }}`, defined in `${KAYOBE_CONFIG_PATH}/inspector.yml`.

kolla_bifrost_ipa_kernel_checksum_url URL of checksum of Ironic Python Agent (IPA) kernel image. Default is `{{ inspector_ipa_kernel_checksum_url }}`, defined in `${KAYOBE_CONFIG_PATH}/inspector.yml`.

kolla_bifrost_ipa_kernel_checksum_algorithm Algorithm of checksum of Ironic Python Agent (IPA) kernel image. Default is `{{ inspector_ipa_kernel_checksum_algorithm }}`, defined in `${KAYOBE_CONFIG_PATH}/inspector.yml`.

kolla_bifrost_ipa_ramdisk_upstream_url URL of Ironic Python Agent (IPA) ramdisk image. Default is `{{ inspector_ipa_ramdisk_upstream_url }}`, defined in `${KAYOBE_CONFIG_PATH}/inspector.yml`.

kolla_bifrost_ipa_ramdisk_checksum_url URL of checksum of Ironic Python Agent (IPA) ramdisk image. Default is `{{ inspector_ipa_ramdisk_checksum_url }}`, defined in `${KAYOBE_CONFIG_PATH}/inspector.yml`.

kolla_bifrost_ipa_ramdisk_checksum_algorithm Algorithm of checksum of Ironic Python Agent (IPA) ramdisk image. Default is `{{ inspector_ipa_ramdisk_checksum_algorithm }}`, defined in `${KAYOBE_CONFIG_PATH}/inspector.yml`.

Inventory configuration

Note: This feature is currently not well tested. It is advisable to use autodiscovery of overcloud servers instead.

The following options are used to configure a static inventory of servers for Bifrost.

`kolla_bifrost_servers`

Server inventory for Bifrost in the JSON file format.

Custom Configuration

Further configuration of arbitrary Ansible variables for Bifrost can be provided via the following files:

- `${KAYOBE_CONFIG_PATH}/kolla/config/bifrost/bifrost.yml`
- `${KAYOBE_CONFIG_PATH}/kolla/config/bifrost/dib.yml`

These are both passed as extra variables files to `ansible-playbook`, but the naming scheme provides a separation of DIB image related variables from other variables. It may be necessary to inspect the Bifrost source code for the full set of variables that may be configured.

For example, to configure debug logging for Ironic Inspector:

Listing 101: `kolla/config/bifrost/bifrost.yml`

```
inspector_debug: true
```

Ironic Python Agent (IPA)

This section covers configuration of Ironic Python Agent (IPA) which is used by Ironic and Ironic Inspector to deploy and inspect bare metal nodes. This is used by the Bifrost services that run on the seed host, and also by Ironic and Ironic Inspector services running in the overcloud for bare metal compute, if enabled (`kolla_enable_ironic` is `true`). IPA configuration is typically applied in `${KAYOBE_CONFIG_PATH}/ipa.yml`. Consult the IPA documentation for full details of IPA usage and configuration.

Ironic Python Agent (IPA) image build configuration

Note: This section may be skipped if not building IPA images locally (`ipa_build_images` is `false`).

The following options cover building of IPA images via Diskimage-builder (DIB). Consult the Diskimage-builder documentation for full details.

The default configuration builds a CentOS 8 ramdisk image which includes the upstream IPA source code, and has a serial console enabled.

The images are built for Bifrost via `kayobe seed deployment image build`, and for Ironic in the overcloud (if enabled) via `kayobe overcloud deployment image build`.

`ipa_build_images` Whether to build IPA images from source. Default is `False`.

`ipa_build_source_url` URL of IPA source repository. Default is https://opendev.org/openstack/ironic-python-agent

`ipa_build_source_version` Version of IPA source repository. Default is `{{ openstack_branch }}`.

`ipa_builder_source_url` URL of IPA builder source repository. Default is https://opendev.org/openstack/ironic-python-agent-builder

`ipa_builder_source_version` Version of IPA builder source repository. Default is `master`.

`ipa_build_dib_elements_default` List of default Diskimage Builder (DIB) elements to use when building IPA images. Default is `["centos", "enable-serial-console", "ironic-python-agent-ramdisk"]`.

`ipa_build_dib_elements_extra` List of additional Diskimage Builder (DIB) elements to use when building IPA images. Default is empty.

`ipa_build_dib_elements` List of Diskimage Builder (DIB) elements to use when building IPA images. Default is combination of `ipa_build_dib_elements_default` and `ipa_build_dib_elements_extra`.

`ipa_build_dib_env_default` Dictionary of default environment variables to provide to Diskimage Builder (DIB) during IPA image build. Default is `{"DIB_RELEASE": "8-stream", "DIB_REPOLOCATION_ironic_python_agent": "{{ ipa_build_source_url }}", "DIB_REPOREF_ironic_python_agent": "{{ ipa_build_source_version }}", "DIB_REPOREF_requirements": "{{ openstack_branch }}"}`.

`ipa_build_dib_env_extra` Dictionary of additional environment variables to provide to Diskimage Builder (DIB) during IPA image build. Default is empty.

`ipa_build_dib_env` Dictionary of environment variables to provide to Diskimage Builder (DIB) during IPA image build. Default is a combination of `ipa_build_dib_env_default` and `ipa_build_dib_env_extra`.

`ipa_build_dib_git_elements_default` List of default git repositories containing Diskimage Builder (DIB) elements. See stackhpc.os-images role for usage. Default is one item for IPA builder.

ipa_build_dib_git_elements_extra List of additional git repositories containing Diskimage Builder (DIB) elements. See stackhpc.os-images role for usage. Default is none.

ipa_build_dib_git_elements List of git repositories containing Diskimage Builder (DIB) elements. See stackhpc.os-images role for usage. Default is combination of `ipa_build_dib_git_elements_default` and `ipa_build_dib_git_elements_extra`.

ipa_build_dib_packages List of DIB packages to install. Default is none.

ipa_build_upper_constraints_file Upper constraints file for installing packages in the virtual environment used for building IPA images. To build CentOS Stream 8 images, default is `https://releases.openstack.org/constraints/upper/wallaby`.

Example: Building IPA images locally

To build IPA images locally:

Listing 102: `ipa.yml`

```
ipa_build_images: true
```

Example: Installing IPA from a custom git repository

To install IPA from a custom git repository:

Listing 103: `ipa.yml`

```
ipa_source_url: https://git.example.com/ironic-python-agent
ipa_source_version: downstream
```

Example: Adding an element

In the following example, we extend the list of DIB elements to add the mellanox element, which can be useful for inspecting hardware with Mellanox InfiniBand NICs.

Listing 104: ipa.yml

```
ipa_build_dib_elements_extra:
  - "mellanox"
```

Example: Configuring a development user account

Warning: A development user account should not be used in production.

When debugging a failed deployment, it can sometimes be necessary to allow access to the image via a preconfigured user account with a known password. This can be achieved via the devuser element.

This example shows how to add the devuser element, and configure a username and password for an account that has passwordless sudo:

Listing 105: ipa.yml

```
ipa_build_dib_elements_extra:
  - "devuser"

ipa_build_dib_env_extra:
  DIB_DEV_USER_USERNAME: "devuser"
  DIB_DEV_USER_PASSWORD: "correct horse battery staple"
  DIB_DEV_USER_PWDLESS_SUDO: "yes"
```

Alternatively, the dynamic-login element can be used to authorize SSH keys by appending them to the kernel arguments.

Further information on troubleshooting IPA can be found here.

Example: Configuring custom DIB elements

Sometimes it is useful to use custom DIB elements that are not shipped with DIB itself. This can be done by sharing them in a git repository.

Listing 106: ipa.yml

```
ipa_build_dib_elements_extra:
  - "my-element"

ipa_build_dib_git_elements:
  - repo: "https://git.example.com/custom-dib-elements"
    local: "{{ source_checkout_path }}/custom-dib-elements"
    version: "master"
    elements_path: "elements"
```

In this example the master branch of https://git.example.com/custom-dib-elements would have a top level elements directory, containing a my-element directory for the element.

Example: Installing a package

It can be necessary to install additional packages in the IPA image. Rather than needing to write a custom DIB element, we can use the `ipa_build_dib_packages` variable. For example, to install the `biosdevname` package:

Listing 107: `ipa.yml`

```
ipa_build_dib_packages:
  - "biosdevname"
```

Ironic Python Agent (IPA) images configuration

Note: If building IPA images locally (`ipa_build_images` is `true`) this section can be skipped.

The following options configure the source of Ironic Python Agent images for inspection and deployment. Consult the Ironic Python Agent documentation for full details.

`ipa_images_upstream_url_suffix` Suffix of upstream Ironic deployment image files. Default is based on `{{ openstack_branch }}`.

`ipa_images_kernel_name` Name of Ironic deployment kernel image to register in Glance. Default is `ipa.kernel`.

`ipa_kernel_upstream_url` URL of Ironic deployment kernel image to download. Default is `https://tarballs.openstack.org/ironic-python-agent/dib/files/ipa-centos8{{ ipa_images_upstream_url_suffix }}.kernel`.

`ipa_kernel_checksum_url` URL of checksum of Ironic deployment kernel image. Default is `{{ ipa_kernel_upstream_url }}.{{ ipa_kernel_checksum_algorithm }}`.

`ipa_kernel_checksum_algorithm` Algorithm of checksum of Ironic deployment kernel image. Default is `sha256`.

`ipa_images_ramdisk_name` Name of Ironic deployment ramdisk image to register in Glance. Default is `ipa.initramfs`.

`ipa_ramdisk_upstream_url` URL of Ironic deployment ramdisk image to download. Default is `https://tarballs.openstack.org/ironic-python-agent/dib/files/ipa-centos8{{ ipa_images_upstream_url_suffix }}.initramfs`.

`ipa_ramdisk_checksum_url` URL of checksum of Ironic deployment ramdisk image. Default is `{{ ipa_ramdisk_upstream_url }}.{{ ipa_ramdisk_checksum_algorithm }}`.

`ipa_ramdisk_checksum_algorithm` Algorithm of checksum of Ironic deployment ramdisk image. Default is `sha256`.

Ironic Python Agent (IPA) deployment configuration

The following options configure how IPA operates during deployment and inspection.

ipa_collect_lldp Whether to enable collection of LLDP TLVs. Default is `True`.

ipa_collectors_default

> **Note:** `extra-hardware` is not currently included as it requires a ramdisk with the `hardware` python module installed.
>
> List of default inspection collectors to run. Default is `["default", "logs", "pci-devices"]`.

ipa_collectors_extra List of additional inspection collectors to run. Default is none.

ipa_collectors List of inspection collectors to run. Default is a combination of `ipa_collectors_default` and `ipa_collectors_extra`.

ipa_benchmarks_default List of default inspection benchmarks to run. Default is `["cpu", "disk", "ram"]`.

ipa_benchmarks_extra List of extra inspection benchmarks to run. Default is none.

ipa_benchmarks

> **Note:** The `extra-hardware` collector must be enabled in order to execute benchmarks during inspection.
>
> List of inspection benchmarks to run. Default is a combination of `ipa_benchmarks_default` and `ipa_benchmarks_extra`.

ipa_kernel_options_default List of default kernel parameters for Ironic python agent. Default includes `ipa-collect-lldp`, `ipa-inspection-collectors` and `ipa-inspection-benchmarks`, with arguments taken from `ipa_collect_lldp`, `ipa_collectors` and `ipa_benchmarks`.

ipa_kernel_options_extra List of additional kernel parameters for Ironic python agent. Default is none.

ipa_kernel_options List of kernel parameters for Ironic python agent. Default is a combination of `ipa_kernel_options_default` and `ipa_kernel_options_extra`.

Example: Adding the `extra-hardware` collector

The `extra-hardware` collector may be used to collect additional information about hardware during inspection. It is also a requirement for running benchmarks. This collector depends on the Python hardware package, which is not installed in IPA images by default.

The following example enables the `extra-hardware` collector:

Listing 108: `ipa.yml`

```
ipa_collectors_extra:
  - "extra-hardware"
```

The `ironic-python-agent-builder` repository provides an extra-hardware element which may be used to install this package. It may be used as follows if building an IPA image locally:

Listing 109: `ipa.yml`

```
ipa_build_dib_elements_extra:
  - "extra-hardware"
```

Example: Passing additional kernel arguments to IPA

The following example shows how to pass additional kernel arguments to IPA:

Listing 110: `ipa.yml`

```
ipa_kernel_options_extra:
  - "foo=bar"
```

Docker registry

This section covers configuration of the Docker registry that may be deployed, by default on the seed host. Docker registry configuration is typically applied in `${KAYOBE_CONFIG_PATH}/docker-registry.yml`. Consult the Docker registry documentation for further details of registry usage and configuration.

The registry is deployed during the `kayobe seed host configure` command.

Configuring the registry

`docker_registry_enabled` Whether a docker registry is enabled. Default is `false`. When set to `true`, the Docker registry is deployed on all hosts in the `docker-registry` group. By default this includes the seed host.

`docker_registry_env` Dict of environment variables to provide to the docker registry container. This allows to configure the registry by overriding specific configuration options, as described at https://docs.docker.com/registry/configuration/ For example, the registry can be configured as a pull through cache to Docker Hub by setting REGISTRY_PROXY_REMOTEURL to https://registry-1.docker.io. Note that it is not possible to push to a registry configured as a pull through cache. Default is `{}`.

`docker_registry_port` The port on which the docker registry server should listen. Default is 4000.

`docker_registry_datadir_volume` Name or path to use as the volume for the docker registry. Default is `docker_registry`.

TLS

It is recommended to enable TLS for the registry.

docker_registry_enable_tls Whether to enable TLS for the registry. Default is `false`.

docker_registry_cert_path Path to a TLS certificate to use when TLS is enabled. Default is none.

docker_registry_key_path Path to a TLS key to use when TLS is enabled. Default is none.

For example, the certificate and key could be stored with the Kayobe configuration, under `${KAYOBE_CONFIG_PATH}/docker-registry/`. These files may be encrypted via Ansible Vault.

Listing 111: `docker-registry.yml`

```
docker_registry_enable_tls: true
docker_registry_cert_path: "{{ kayobe_config_path }}/docker-registry/cert.
↪pem"
docker_registry_key_path: "{{ kayobe_config_path }}/docker-registry/key.pem
↪"
```

Basic authentication

It is recommended to enable HTTP basic authentication for the registry. This needs to be done in conjunction with enabling TLS for the registry: using basic authentication over unencrypted HTTP is not supported.

docker_registry_enable_basic_auth Whether to enable basic authentication for the registry. Default is `false`.

docker_registry_basic_auth_htpasswd_path Path to a htpasswd formatted password store for the registry. Default is none.

The password store uses a `htpasswd` format. The following example shows how to generate a password and add it to the `kolla` user in the password store. The password store may be stored with the Kayobe configuration, under `${KAYOBE_CONFIG_PATH}/docker-registry/`. The file may be encrypted via Ansible Vault.

```
uuidgen | tr -d '\n' > registry-password
cat registry-password | docker run --rm -i --entrypoint htpasswd␣
↪httpd:latest -niB kolla > $KAYOBE_CONFIG_PATH/docker-registry/htpasswd
```

Next we configure Kayobe to enable basic authentication for the registry, and specify the path to the password store.

Listing 112: `docker-registry.yml`

```
docker_registry_enable_basic_auth: true
docker_registry_basic_auth_htpasswd_path: "{{ kayobe_config_path }}/docker-
↪registry/htpasswd"
```

Using the registry

Enabling the registry does not automatically set the configuration for Docker engine to use it. This should be done via the *docker_registry variable*.

TLS

If the registry is using a privately signed TLS certificate, it is necessary to *configure Docker engine with the CA certificate*.

If TLS is enabled, Docker engine should be configured to use HTTPS to communicate with it:

Listing 113: `kolla/globals.yml`

```
docker_registry_insecure: false
```

Basic authentication

If basic authentication is enabled, Kolla Ansible needs to be configured with the username and password.

Listing 114: `kolla.yml`

```
kolla_docker_registry_username: <registry username>
kolla_docker_registry_password: <registry password>
```

Seed custom containers

This section covers configuration of the user-defined containers deployment functionality that runs on the seed host.

Configuration

For example, to deploy a squid container image:

Listing 115: `seed.yml`

```
seed_containers:
  squid:
    image: "stackhpc/squid:3.5.20-1"
    pre: "{{ kayobe_config_path }}/containers/squid/pre.yml"
    post: "{{ kayobe_config_path }}/containers/squid/post.yml"
```

Please notice the *optional* pre and post Ansible task files - those need to be created in `kayobe-config` path and will be run before and after particular container deployment.

Possible options for container deployment:

```
seed_containers:
  containerA:
    capabilities:
    command:
    comparisons:
    detach:
    env:
    network_mode:
    image:
    init:
    ipc_mode:
    pid_mode:
    ports:
    privileged:
    restart_policy:
    sysctls:
    tag:
    ulimits:
    user:
    volumes:
```

For a detailed explanation of each option - please see Ansible docker_container module page.

List of Kayobe applied defaults to required docker_container variables:

```
---
deploy_containers_defaults:
  comparisons:
    image: strict
    env: strict
    volumes: strict
  detach: True
  network_mode: "host"
  init: True
  privileged: False
  restart_policy: "unless-stopped"

deploy_containers_docker_api_timeout: 120
```

Nova cells

In the Train release, Kolla Ansible gained full support for the Nova cells v2 scale out feature. Whilst configuring Nova cells is documented in Kolla Ansible, implementing that configuration in Kayobe is documented here.

In Kolla Ansible, Nova cells are configured via group variables. In Kayobe, these group variables can be set via Kayobe configuration. For example, to configure `cell0001` the following file could be created:

Listing 116: $KAYOBE_CONFIG_PATH/kolla/inventory/group_vars/cell0001/all

```
---
nova_cell_name: cell0001
nova_cell_novncproxy_group: cell0001-vnc
nova_cell_conductor_group: cell0001-control
nova_cell_compute_group: cell0001-compute
```

After defining the cell `group_vars` the Kayobe inventory can be configured. In Kayobe, cell controllers and cell compute hosts become part of the existing `controllers` and `compute` Kayobe groups because typically they will need to be provisioned in the same way. In Kolla Ansible, to prevent non-cell services being mapped to cell controllers, the `controllers` group must be split into two. The inventory file should also include the cell definitions. The following groups and hosts files give an example of how this may be achieved:

Listing 117: $KAYOBE_CONFIG_PATH/inventory/groups

```
# Kayobe groups inventory file. This file should generally not be␣
↪modified.
# If declares the top-level groups and sub-groups.

##############################################################################
↪#####
# Seed groups.

[seed]
# Empty group to provide declaration of seed group.

[seed-hypervisor]
# Empty group to provide declaration of seed-hypervisor group.

[container-image-builders:children]
# Build container images on the seed by default.
seed

##############################################################################
↪#####
# Overcloud groups.

[controllers]
# Empty group to provide declaration of controllers group.

[network:children]
# Add controllers to network group by default for backwards compatibility,
# although they could be separate hosts.
top-level-controllers

[monitoring]
# Empty group to provide declaration of monitoring group.

[storage]
# Empty group to provide declaration of storage group.
```

(continues on next page)

(continued from previous page)

```
[compute]
# Empty group to provide declaration of compute group.

# Empty group to provide declaration of top-level controllers.
[top-level-controllers]

[overcloud:children]
controllers
network
monitoring
storage
compute

##############################################################################
# Docker groups.

[docker:children]
# Hosts in this group will have Docker installed.
seed
controllers
network
monitoring
storage
compute

[docker-registry:children]
# Hosts in this group will have a Docker Registry deployed. This group should
# generally contain only a single host, to avoid deploying multiple independent
# registries which may become unsynchronized.
seed

##############################################################################
# Baremetal compute node groups.

[baremetal-compute]
# Empty group to provide declaration of baremetal-compute group.

##############################################################################
# Networking groups.

[mgmt-switches]
# Empty group to provide declaration of mgmt-switches group.

[ctl-switches]
# Empty group to provide declaration of ctl-switches group.

[hs-switches]
# Empty group to provide declaration of hs-switches group.

[switches:children]
mgmt-switches
```

(continues on next page)

(continued from previous page)

```
ctl-switches
hs-switches
```

Listing 118: `$KAYOBE_CONFIG_PATH/inventory/hosts`

```
# Kayobe hosts inventory file. This file should be modified to define the␣
↪hosts
# and their top-level group membership.

# This host acts as the configuration management Ansible control host.␣
↪This must be
# localhost.
localhost ansible_connection=local

[seed-hypervisor]
# Add a seed hypervisor node here if required. This host will run a seed␣
↪node
# Virtual Machine.

[seed]
operator

[controllers:children]
top-level-controllers
cell-controllers

[top-level-controllers]
control01

[cell-controllers:children]
cell01-control
cell02-control

[compute:children]
cell01-compute
cell02-compute

[cell01:children]
cell01-control
cell01-compute
cell01-vnc

[cell01-control]
control02

[cell01-vnc]
control02

[cell01-compute]
compute01

[cell02:children]
cell02-control
cell02-compute
cell02-vnc
```

(continues on next page)

(continued from previous page)

```
[cell02-control]
control03

[cell02-vnc]
control03

[cell02-compute]
compute02
compute03

###################################

[mgmt-switches]
# Add management network switches here if required.

[ctl-switches]
# Add control and provisioning switches here if required.

[hs-switches]
# Add high speed switches here if required.
```

Having configured the Kayobe inventory, the Kolla Ansible inventory can be configured. Currently this can be done via the `kolla_overcloud_inventory_top_level_group_map` variable. For example, to configure the two cells defined in the Kayobe inventory above, the variable could be set to the following:

Listing 119: `$KAYOBE_CONFIG_PATH/kolla.yml`

```
kolla_overcloud_inventory_top_level_group_map:
  control:
    groups:
      - top-level-controllers
  network:
    groups:
      - network
  compute:
    groups:
      - compute
  monitoring:
    groups:
      - monitoring
  cell-control:
    groups:
      - cell-controllers
  cell0001:
    groups:
      - cell01
  cell0001-control:
     groups:
      - cell01-control
  cell0001-compute:
    groups:
      - cell01-compute
  cell0001-vnc:
```

(continues on next page)

(continued from previous page)

```
    groups:
      - cell01-vnc
  cell0002:
    groups:
      - cell02
  cell0002-control:
    groups:
      - cell02-control
  cell0002-compute:
    groups:
      - cell02-compute
  cell0002-vnc:
    groups:
      - cell02-vnc
```

Finally, Nova cells can be enabled in Kolla Ansible:

Listing 120: `$KAYOBE_CONFIG_PATH/kolla/globals.yml`

```
enable_cells: True
```

3.7 Deployment

This section describes usage of Kayobe to install an OpenStack cloud onto a set of bare metal servers. We assume access is available to a node which will act as the hypervisor hosting the seed node in a VM. We also assume that this seed hypervisor has access to the bare metal nodes that will form the OpenStack control plane. Finally, we assume that the control plane nodes have access to the bare metal nodes that will form the workload node pool.

See also:

Information on the configuration of a Kayobe environment is available *here*.

3.7.1 Ansible Control Host

Before starting deployment we must bootstrap the Ansible control host. Tasks performed here include:

- Install Ansible and role dependencies from Ansible Galaxy.
- Generate an SSH key if necessary and add it to the current users authorised keys.
- Install Kolla Ansible locally at the configured version.

To bootstrap the Ansible control host:

```
(kayobe) $ kayobe control host bootstrap
```

3.7.2 Physical Network

The physical network can be managed by Kayobe, which uses Ansibles network modules. Currently Dell Network OS 6 and Dell Network OS 9 switches are supported but this could easily be extended. To provision the physical network:

```
(kayobe) $ kayobe physical network configure --group <group> [--enable-
↪discovery]
```

The `--group` argument is used to specify an Ansible group containing the switches to be configured.

The `--enable-discovery` argument enables a one-time configuration of ports attached to baremetal compute nodes to support hardware discovery via ironic inspector.

It is possible to limit the switch interfaces that will be configured, either by interface name or interface description:

```
(kayobe) $ kayobe physical network configure --group <group> --interface-
↪limit <interface names>
(kayobe) $ kayobe physical network configure --group <group> --interface-
↪description-limit <interface descriptions>
```

The names or descriptions should be separated by commas. This may be useful when adding compute nodes to an existing deployment, in order to avoid changing the configuration interfaces in use by active nodes.

The `--display` argument will display the candidate switch configuration, without actually applying it.

See also:

Information on configuration of physical network devices is available *here*.

3.7.3 Seed Hypervisor

Note: It is not necessary to run the seed services in a VM. To use an existing bare metal host or a VM provisioned outside of Kayobe, this section may be skipped.

Host Configuration

To configure the seed hypervisors host OS, and the Libvirt/KVM virtualisation support:

```
(kayobe) $ kayobe seed hypervisor host configure
```

See also:

Information on configuration of hosts is available *here*.

3.7.4 Seed

VM Provisioning

Note: It is not necessary to run the seed services in a VM. To use an existing bare metal host or a VM provisioned outside of Kayobe, this step may be skipped. Ensure that the Ansible inventory contains a host for the seed.

The seed hypervisor should have CentOS and `libvirt` installed. It should have `libvirt` networks configured for all networks that the seed VM needs access to and a `libvirt` storage pool available for the seed VMs volumes. To provision the seed VM:

```
(kayobe) $ kayobe seed vm provision
```

When this command has completed the seed VM should be active and accessible via SSH. Kayobe will update the Ansible inventory with the IP address of the VM.

Host Configuration

To configure the seed host OS:

```
(kayobe) $ kayobe seed host configure
```

Note: If the seed host uses disks that have been in use in a previous installation, it may be necessary to wipe partition and LVM data from those disks. To wipe all disks that are not mounted during host configuration:

```
(kayobe) $ kayobe seed host configure --wipe-disks
```

See also:

Information on configuration of hosts is available *here*.

Building Container Images

Note: It is possible to use prebuilt container images from an image registry such as Dockerhub. In this case, this step can be skipped.

It is possible to use prebuilt container images from an image registry such as Dockerhub. In some cases it may be necessary to build images locally either to apply local image customisation or to use a downstream version of kolla. Images are built by hosts in the `container-image-builders` group, which by default includes the `seed`.

To build container images:

```
(kayobe) $ kayobe seed container image build
```

It is possible to build a specific set of images by supplying one or more image name regular expressions:

```
(kayobe) $ kayobe seed container image build bifrost-deploy
```

In order to push images to a registry after they are built, add the `--push` argument.

See also:

Information on configuration of Kolla for building container images is available *here*.

Deploying Containerised Services

At this point the seed services need to be deployed on the seed VM. These services are deployed in the `bifrost_deploy` container.

This command will also build the Operating System image that will be used to deploy the overcloud nodes using Disk Image Builder (DIB).

To deploy the seed services in containers:

```
(kayobe) $ kayobe seed service deploy
```

After this command has completed the seed services will be active.

See also:

Information on configuration of Kolla Ansible is available *here*. See *here* for information about configuring Bifrost. *Overcloud root disk image configuration* provides information on configuring the root disk image build process. See *here* for information about deploying additional, custom containers on seed node.

Building Deployment Images

Note: It is possible to use prebuilt deployment images. In this case, this step can be skipped.

It is possible to use prebuilt deployment images from the OpenStack hosted tarballs or another source. In some cases it may be necessary to build images locally either to apply local image customisation or to use a downstream version of Ironic Python Agent (IPA). In order to build IPA images, the `ipa_build_images` variable should be set to `True`.

To build images locally:

```
(kayobe) $ kayobe seed deployment image build
```

If images have been built previously, they will not be rebuilt. To force rebuilding images, use the `--force-rebuild` argument.

See also:

See *here* for information on how to configure the IPA image build process.

Accessing the Seed via SSH (Optional)

For SSH access to the seed, first determine the seeds IP address. We can use the `kayobe configuration dump` command to inspect the seeds IP address:

```
(kayobe) $ kayobe configuration dump --host seed --var-name ansible_host
```

The `kayobe_ansible_user` variable determines which user account will be used by Kayobe when accessing the machine via SSH. By default this is `stack`. Use this user to access the seed:

```
$ ssh <kayobe ansible user>@<seed VM IP>
```

To see the active Docker containers:

```
$ docker ps
```

Leave the seed VM and return to the shell on the Ansible control host:

```
$ exit
```

3.7.5 Overcloud

Discovery

Note: If discovery of the overcloud is not possible, a static inventory of servers using the bifrost `servers.yml` file format may be configured using the `kolla_bifrost_servers` variable in `${KAYOBE_CONFIG_PATH}/bifrost.yml`.

Discovery of the overcloud is supported by the ironic inspector service running in the `bifrost_deploy` container on the seed. The service is configured to PXE boot unrecognised MAC addresses with an IPA ramdisk for introspection. If an introspected node does not exist in the ironic inventory, ironic inspector will create a new entry for it.

Discovery of the overcloud is triggered by causing the nodes to PXE boot using a NIC attached to the overcloud provisioning network. For many servers this will be the factory default and can be performed by powering them on.

On completion of the discovery process, the overcloud nodes should be registered with the ironic service running in the seed hosts `bifrost_deploy` container. The node inventory can be viewed by executing the following on the seed:

```
$ docker exec -it bifrost_deploy bash
(bifrost_deploy) $ export OS_CLOUD=bifrost
(bifrost_deploy) $ baremetal node list
```

In order to interact with these nodes using Kayobe, run the following command to add them to the Kayobe and Kolla-Ansible inventories:

```
(kayobe) $ kayobe overcloud inventory discover
```

See also:

This blog post provides a case study of the discovery process, including automatically naming Ironic nodes via switch port descriptions, Ironic Inspector and LLDP.

Saving Hardware Introspection Data

If ironic inspector is in use on the seed host, introspection data will be stored in the local nginx service. This data may be saved to the control host:

```
(kayobe) $ kayobe overcloud introspection data save
```

`--output-dir` may be used to specify the directory in which introspection data files will be saved. `--output-format` may be used to set the format of the files.

BIOS and RAID Configuration

Note: BIOS and RAID configuration may require one or more power cycles of the hardware to complete the operation. These will be performed automatically.

Note: Currently, BIOS and RAID configuration of overcloud hosts is supported for Dell servers only.

Configuration of BIOS settings and RAID volumes is currently performed out of band as a separate task from hardware provisioning. To configure the BIOS and RAID:

```
(kayobe) $ kayobe overcloud bios raid configure
```

After configuring the nodes RAID volumes it may be necessary to perform hardware inspection of the nodes to reconfigure the ironic nodes scheduling properties and root device hints. To perform manual hardware inspection:

```
(kayobe) $ kayobe overcloud hardware inspect
```

There are currently a few limitations to configuring BIOS and RAID:

- The Ansible control host must be able to access the BMCs of the servers being configured.
- The Ansible control host must have the `python-dracclient` Python module available to the Python interpreter used by Ansible. The path to the Python interpreter is configured via `ansible_python_interpreter`.

Provisioning

Note: There is a cloud-init issue which prevents Ironic nodes without names from being accessed via SSH after provisioning. To avoid this issue, ensure that all Ironic nodes in the Bifrost inventory are named. This may be achieved via *autodiscovery*, or manually, e.g. from the seed:

```
$ docker exec -it bifrost_deploy bash
(bifrost_deploy) $ export OS_CLOUD=bifrost
(bifrost_deploy) $ baremetal node set ee77b4ca-8860-4003-a18f-b00d01295bda␣
↪--name controller0
```

Provisioning of the overcloud is performed by the ironic service running in the bifrost container on the seed. To provision the overcloud nodes:

```
(kayobe) $ kayobe overcloud provision
```

After this command has completed the overcloud nodes should have been provisioned with an OS image. The command will wait for the nodes to become `active` in ironic and accessible via SSH.

Host Configuration

To configure the overcloud hosts OS:

```
(kayobe) $ kayobe overcloud host configure
```

Note: If the controller hosts use disks that have been in use in a previous installation, it may be necessary to wipe partition and LVM data from those disks. To wipe all disks that are not mounted during host configuration:

```
(kayobe) $ kayobe overcloud host configure --wipe-disks
```

See also:

Information on configuration of hosts is available *here*.

Building Container Images

Note: It is possible to use prebuilt container images from an image registry such as Dockerhub. In this case, this step can be skipped.

In some cases it may be necessary to build images locally either to apply local image customisation or to use a downstream version of kolla. Images are built by hosts in the `container-image-builders` group, which by default includes the `seed`. If no seed host is in use, for example in an all-in-one controller development environment, this group may be modified to cause containers to be built on the controllers.

To build container images:

```
(kayobe) $ kayobe overcloud container image build
```

It is possible to build a specific set of images by supplying one or more image name regular expressions:

```
(kayobe) $ kayobe overcloud container image build ironic- nova-api
```

In order to push images to a registry after they are built, add the `--push` argument.

See also:

Information on configuration of Kolla for building container images is available *here*.

Pulling Container Images

Note: It is possible to build container images locally avoiding the need for an image registry such as Dockerhub. In this case, this step can be skipped.

In most cases suitable prebuilt kolla images will be available on Dockerhub. The kolla account provides image repositories suitable for use with kayobe and will be used by default. To pull images from the configured image registry:

```
(kayobe) $ kayobe overcloud container image pull
```

Building Deployment Images

Note: It is possible to use prebuilt deployment images. In this case, this step can be skipped.

Note: Deployment images are only required for the overcloud when Ironic is in use. Otherwise, this step can be skipped.

It is possible to use prebuilt deployment images from the OpenStack hosted tarballs or another source. In some cases it may be necessary to build images locally either to apply local image customisation or to use a downstream version of Ironic Python Agent (IPA). In order to build IPA images, the `ipa_build_images` variable should be set to `True`.

To build images locally:

```
(kayobe) $ kayobe overcloud deployment image build
```

If images have been built previously, they will not be rebuilt. To force rebuilding images, use the `--force-rebuild` argument.

See also:

See *here* for information on how to configure the IPA image build process.

Building Swift Rings

Note: This section can be skipped if Swift is not in use.

Swift uses ring files to control placement of data across a cluster. These files can be generated automatically using the following command:

```
(kayobe) $ kayobe overcloud swift rings generate
```

Deploying Containerised Services

To deploy the overcloud services in containers:

```
(kayobe) $ kayobe overcloud service deploy
```

Once this command has completed the overcloud nodes should have OpenStack services running in Docker containers.

See also:

Information on configuration of Kolla Ansible is available *here*.

Interacting with the Control Plane

Kolla-ansible writes out an environment file that can be used to access the OpenStack admin endpoints as the admin user:

```
$ source ${KOLLA_CONFIG_PATH:-/etc/kolla}/admin-openrc.sh
```

Kayobe also generates an environment file that can be used to access the OpenStack public endpoints as the admin user which may be required if the admin endpoints are not available from the Ansible control host:

```
$ source ${KOLLA_CONFIG_PATH:-/etc/kolla}/public-openrc.sh
```

Performing Post-deployment Configuration

To perform post deployment configuration of the overcloud services:

```
(kayobe) $ source ${KOLLA_CONFIG_PATH:-/etc/kolla}/admin-openrc.sh
(kayobe) $ kayobe overcloud post configure
```

This will perform the following tasks:

- Register Ironic Python Agent (IPA) images with glance
- Register introspection rules with ironic inspector
- Register a provisioning network and subnet with neutron
- Configure Grafana organisations, dashboards and datasources

3.8 Upgrading

This section describes how to upgrade from one OpenStack release to another.

3.8.1 Preparation

Before you start, be sure to back up any local changes, configuration, and data.

Migrating Kayobe Configuration

Kayobe configuration options may be changed between releases of kayobe. Ensure that all site local configuration is migrated to the target version format. If using the kayobe-config git repository to manage local configuration, this process can be managed via git. For example, to fetch version 1.0.0 of the configuration from the `origin` remote and merge it into the current branch:

```
$ git fetch origin 1.0.0
$ git merge 1.0.0
```

The configuration should be manually inspected after the merge to ensure that it is correct. Any new configuration options may be set at this point. In particular, the following options may need to be changed if not using their default values:

- `kolla_openstack_release`
- `kolla_tag`
- `kolla_sources`
- `kolla_build_blocks`
- `kolla_build_customizations`

Once the configuration has been migrated, it is possible to view the global variables for all hosts:

```
(kayobe) $ kayobe configuration dump
```

The output of this command is a JSON object mapping hosts to their configuration. The output of the command may be restricted using the `--host`, `--hosts`, `--var-name` and `--dump-facts` options.

If using the `kayobe-env` environment file in `kayobe-config`, this should also be inspected for changes and modified to suit the local ansible control host environment if necessary. When ready, source the environment file:

```
$ source kayobe-env
```

The Kayobe release notes provide information on each new release. In particular, the *Upgrade Notes* and *Deprecation Notes* sections provide information that might affect the configuration migration.

All changes made to the configuration should be committed and pushed to the hosting git repository.

3.8.2 Updating Kayobe Configuration

Ensure that the Kayobe configuration is checked out at the required commit.

First, ensure that there are no uncommitted local changes to the repository:

```
$ cd <base_path>/src/kayobe-config/
$ git status
```

Pull down changes from the hosting repository. For example, to fetch changes from the `master` branch of the `origin` remote:

```
$ git checkout master
$ git pull --ff-only origin master
```

Adjust this procedure to suit your environment.

3.8.3 Upgrading Kayobe

If a new, suitable version of kayobe is available, it should be installed. As described in *Installation*, Kayobe can be installed via the released Python packages on PyPI, or from source. Installation from a Python package is supported from Kayobe 5.0.0 onwards.

Upgrading from PyPI

This section describes how to upgrade Kayobe from a Python package in a virtualenv. This is supported from Kayobe 5.0.0 onwards.

Ensure that the virtualenv is activated:

```
$ source <base_path>/venvs/kayobe/bin/activate
```

Update the pip package:

```
(kayobe) $ pip install -U pip
```

If upgrading to the latest version of Kayobe:

```
(kayobe) $ pip install -U kayobe
```

Alternatively, to upgrade to a specific release of Kayobe:

```
(kayobe) $ pip install kayobe==5.0.0
```

Upgrading from source

This section describes how to install Kayobe from source in a virtualenv.

First, check out the required version of the Kayobe source code. This may be done by pulling down the new version from `opendev.org`. Make sure that any local changes to kayobe are committed and merged with the new upstream code as necessary. For example, to pull version 5.0.0 from the `origin` remote:

```
$ cd <base_path>/src/kayobe
$ git pull origin 5.0.0
```

Ensure that the virtualenv is activated:

```
$ source <base_path>/venvs/kayobe/bin/activate
```

Update the pip package:

```
(kayobe) $ pip install -U pip
```

If using a non-editable install of Kayobe:

```
(kayobe) $ cd <base_path>/src/kayobe
(kayobe) $ pip install -U .
```

Alternatively, if using an editable install of Kayobe (version 5.0.0 onwards, see *Editable source installation* for details):

```
(kayobe) $ cd <base_path>/src/kayobe
(kayobe) $ pip install -U -e .
```

3.8.4 Upgrading the Ansible Control Host

Before starting the upgrade we must upgrade the Ansible control host. Tasks performed here include:

- Install updated Ansible role dependencies from Ansible Galaxy.
- Generate an SSH key if necessary and add it to the current users authorised keys.
- Upgrade Kolla Ansible locally to the configured version.

To upgrade the Ansible control host:

```
(kayobe) $ kayobe control host upgrade
```

3.8.5 Upgrading the Seed Hypervisor

Currently, upgrading the seed hypervisor services is not supported. It may however be necessary to upgrade host packages and some host services.

Upgrading Host Packages

Prior to upgrading the seed hypervisor, it may be desirable to upgrade system packages on the seed hypervisor host.

To update all eligible packages, use *, escaping if necessary:

```
(kayobe) $ kayobe seed hypervisor host package update --packages "*"
```

To only install updates that have been marked security related:

```
(kayobe) $ kayobe seed hypervisor host package update --packages "*" --
↪security
```

Upgrading Host Services

It may be necessary to upgrade some host services:

```
(kayobe) $ kayobe seed hypervisor host upgrade
```

Note that this will not perform full configuration of the host, and will instead perform a targeted upgrade of specific services where necessary.

3.8.6 Upgrading the Seed

The seed services are upgraded in two steps. First, new container images should be obtained either by building them locally or pulling them from an image registry. Second, the seed services should be replaced with new containers created from the new container images.

Upgrading Host Packages

Prior to upgrading the seed, it may be desirable to upgrade system packages on the seed host.

To update all eligible packages, use *, escaping if necessary:

```
(kayobe) $ kayobe seed host package update --packages "*"
```

To only install updates that have been marked security related:

```
(kayobe) $ kayobe seed host package update --packages "*" --security
```

Note that these commands do not affect packages installed in containers, only those installed on the host.

Building Ironic Deployment Images

Note: It is possible to use prebuilt deployment images. In this case, this step can be skipped.

It is possible to use prebuilt deployment images from the OpenStack hosted tarballs or another source. In some cases it may be necessary to build images locally either to apply local image customisation or to use a downstream version of Ironic Python Agent (IPA). In order to build IPA images, the `ipa_build_images` variable should be set to `True`. To build images locally:

```
(kayobe) $ kayobe seed deployment image build
```

To overwrite existing images, add the `--force-rebuild` argument.

Upgrading Host Services

It may be necessary to upgrade some host services:

```
(kayobe) $ kayobe seed host upgrade
```

Note that this will not perform full configuration of the host, and will instead perform a targeted upgrade of specific services where necessary.

Building Container Images

Note: It is possible to use prebuilt container images from an image registry such as Dockerhub. In this case, this step can be skipped.

In some cases it may be necessary to build images locally either to apply local image customisation or to use a downstream version of kolla. To build images locally:

```
(kayobe) $ kayobe seed container image build
```

In order to push images to a registry after they are built, add the `--push` argument.

Upgrading Containerised Services

Containerised seed services may be upgraded by replacing existing containers with new containers using updated images which have been pulled from a registry or built locally.

To upgrade the containerised seed services:

```
(kayobe) $ kayobe seed service upgrade
```

3.8.7 Upgrading the Overcloud

The overcloud services are upgraded in two steps. First, new container images should be obtained either by building them locally or pulling them from an image registry. Second, the overcloud services should be replaced with new containers created from the new container images.

Upgrading Host Packages

Prior to upgrading the OpenStack control plane, it may be desirable to upgrade system packages on the overcloud hosts.

To update all eligible packages, use `*`, escaping if necessary:

```
(kayobe) $ kayobe overcloud host package update --packages "*"
```

To only install updates that have been marked security related:

```
(kayobe) $ kayobe overcloud host package update --packages "*" --security
```

Note that these commands do not affect packages installed in containers, only those installed on the host.

Upgrading Host Services

Prior to upgrading the OpenStack control plane, the overcloud host services should be upgraded:

```
(kayobe) $ kayobe overcloud host upgrade
```

Note that this will not perform full configuration of the host, and will instead perform a targeted upgrade of specific services where necessary.

Building Ironic Deployment Images

Note: It is possible to use prebuilt deployment images. In this case, this step can be skipped.

It is possible to use prebuilt deployment images from the OpenStack hosted tarballs or another source. In some cases it may be necessary to build images locally either to apply local image customisation or to use a downstream version of Ironic Python Agent (IPA). In order to build IPA images, the `ipa_build_images` variable should be set to `True`. To build images locally:

```
(kayobe) $ kayobe overcloud deployment image build
```

To overwrite existing images, add the `--force-rebuild` argument.

Upgrading Ironic Deployment Images

Prior to upgrading the OpenStack control plane you should upgrade the deployment images. If you are using prebuilt images, update the following variables in `etc/kayobe/ipa.yml` accordingly:

- `ipa_kernel_upstream_url`
- `ipa_kernel_checksum_url`
- `ipa_kernel_checksum_algorithm`
- `ipa_ramdisk_upstream_url`
- `ipa_ramdisk_checksum_url`
- `ipa_ramdisk_checksum_algorithm`

Alternatively, you can update the files that the URLs point to. If building the images locally, follow the process outlined in *Building Ironic Deployment Images*.

To get Ironic to use an updated set of overcloud deployment images, you can run:

```
(kayobe) $ kayobe baremetal compute update deployment image
```

This will register the images in Glance and update the `deploy_ramdisk` and `deploy_kernel` properties of the Ironic nodes.

Before rolling out the update to all nodes, it can be useful to test the image on a limited subset. To do this, you can use the `baremetal-compute-limit` option. See *Update Deployment Image* for more details.

Building Container Images

Note: It is possible to use prebuilt container images from an image registry such as Dockerhub. In this case, this step can be skipped.

In some cases it may be necessary to build images locally either to apply local image customisation or to use a downstream version of kolla. To build images locally:

```
(kayobe) $ kayobe overcloud container image build
```

It is possible to build a specific set of images by supplying one or more image name regular expressions:

```
(kayobe) $ kayobe overcloud container image build ironic- nova-api
```

In order to push images to a registry after they are built, add the `--push` argument.

Pulling Container Images

Note: It is possible to build container images locally avoiding the need for an image registry such as Dockerhub. In this case, this step can be skipped.

In most cases suitable prebuilt kolla images will be available on Dockerhub. The kolla account provides image repositories suitable for use with kayobe and will be used by default. To pull images from the configured image registry:

```
(kayobe) $ kayobe overcloud container image pull
```

Saving Overcloud Service Configuration

It is often useful to be able to save the configuration of the control plane services for inspection or comparison with another configuration set prior to a reconfiguration or upgrade. This command will gather and save the control plane configuration for all hosts to the Ansible control host:

```
(kayobe) $ kayobe overcloud service configuration save
```

The default location for the saved configuration is `$PWD/overcloud-config`, but this can be changed via the `output-dir` argument. To gather configuration from a directory other than the default `/etc/kolla`, use the `node-config-dir` argument.

Generating Overcloud Service Configuration

Prior to deploying, reconfiguring, or upgrading a control plane, it may be useful to generate the configuration that will be applied, without actually applying it to the running containers. The configuration should typically be generated in a directory other than the default configuration directory of `/etc/kolla`, to avoid overwriting the active configuration:

```
(kayobe) $ kayobe overcloud service configuration generate --node-config-
↪dir /path/to/generated/config
```

The configuration will be generated remotely on the overcloud hosts in the specified directory, with one subdirectory per container. This command may be followed by `kayobe overcloud service configuration save` to gather the generated configuration to the Ansible control host.

Upgrading Containerised Services

Containerised control plane services may be upgraded by replacing existing containers with new containers using updated images which have been pulled from a registry or built locally.

To upgrade the containerised control plane services:

```
(kayobe) $ kayobe overcloud service upgrade
```

It is possible to specify tags for Kayobe and/or kolla-ansible to restrict the scope of the upgrade:

```
(kayobe) $ kayobe overcloud service upgrade --tags config --kolla-tags␣
↪keystone
```

3.9 Administration

This section describes how to use kayobe to simplify post-deployment administrative tasks.

3.9.1 General Administration

Updating the Control Host

There are several pieces of software and configuration that must be installed and synchronised on the Ansible Control host:

- Kayobe configuration
- Kayobe Python package
- Ansible Galaxy roles
- Kolla Ansible Python package

A change to the configuration may require updating the Kolla Ansible Python package. Updating the Kayobe Python package may require updating the Ansible Galaxy roles. Its not always easy to know which of these are required, so the simplest option is to apply all of the following steps when any of the above are changed.

1. *Update Kayobe configuration* to the required commit
2. *Upgrade the Kayobe Python package* to the required version
3. *Upgrade the Ansible control host* to synchronise the Ansible Galaxy roles and Kolla Ansible Python package.

Running Kayobe Playbooks on Demand

In some situations it may be necessary to run an individual Kayobe playbook. Playbooks are stored in `<kayobe repo>/ansible/*.yml`. To run an arbitrary Kayobe playbook:

```
(kayobe) $ kayobe playbook run <playbook> [<playbook>]
```

Running Kolla-ansible Commands

To execute a kolla-ansible command:

```
(kayobe) $ kayobe kolla ansible run <command>
```

Dumping Kayobe Configuration

The Ansible configuration space is quite large, and it can be hard to determine the final values of Ansible variables. We can use Kayobes `configuration dump` command to view individual variables or the variables for one or more hosts. To dump Kayobe configuration for one or more hosts:

```
(kayobe) $ kayobe configuration dump
```

The output is a JSON-formatted object mapping hosts to their hostvars.

We can use the `--var-name` argument to inspect a particular variable or the `--host` or `--hosts` arguments to view a variable or variables for a specific host or set of hosts.

Checking Network Connectivity

In complex networking environments it can be useful to be able to automatically check network connectivity and diagnose networking issues. To perform some simple connectivity checks:

```
(kayobe) $ kayobe network connectivity check
```

Note that this will run on the seed, seed hypervisor, and overcloud hosts. If any of these hosts are not expected to be active (e.g. prior to overcloud deployment), the set of target hosts may be limited using the `--limit` argument.

3.9.2 Seed Administration

Deprovisioning The Seed VM

Note: This step will destroy the seed VM and its data volumes.

To deprovision the seed VM:

```
(kayobe) $ kayobe seed vm deprovision
```

Updating Packages

It is possible to update packages on the seed host.

Package Repositories

If using custom package repositories, it may be necessary to update these prior to running a package update. To do this, update the configuration in `${KAYOBE_CONFIG_PATH}/dnf.yml` and run the following command:

```
(kayobe) $ kayobe seed host configure --tags dnf --kolla-tags none
```

Package Update

To update one or more packages:

```
(kayobe) $ kayobe seed host package update --packages <package1>,<package2>
```

To update all eligible packages, use *, escaping if necessary:

```
(kayobe) $ kayobe seed host package update --packages "*"
```

To only install updates that have been marked security related:

```
(kayobe) $ kayobe seed host package update --packages "*" --security
```

Note that these commands do not affect packages installed in containers, only those installed on the host.

Packages can also be updated on the seed hypervisor host, if one is in use:

```
(kayobe) $ kayobe seed hypervisor package update --packages <package1>,
↪<package2>
```

Kernel Updates

If the kernel has been updated, you will probably want to reboot the seed host to boot into the new kernel. This can be done using a command such as the following:

```
(kayobe) $ kayobe seed host command run --command "shutdown -r" --become
```

Examining the Bifrost Container

The seed host runs various services required for a standalone Ironic deployment. These all run in a single `bifrost_deploy` container.

It can often be helpful to execute a shell in the bifrost container for diagnosing operational issues:

```
$ docker exec -it bifrost_deploy bash
```

Services are run via Systemd:

```
(bifrost_deploy) systemctl
```

Logs are stored in `/var/log/kolla/`, which is mounted to the `kolla_logs` Docker volume.

Accessing the Seed Services

The Ironic and Ironic inspector APIs can be accessed via the `baremetal` command line interface:

```
(bifrost_deploy) $ export OS_CLOUD=bifrost
(bifrost_deploy) $ baremetal node list
(bifrost_deploy) $ baremetal introspection list
```

Backup & Restore

There are two main approaches to backing up and restoring data on the seed. A backup may be taken of the Ironic databases. Alternatively, a Virtual Machine backup may be used if running the seed services in a VM. The former will consume less storage. Virtual Machine backups are not yet covered here, neither is scheduling of backups. Any backup and restore procedure should be tested in advance.

Database Backup & Restore

A backup may be taken of the database, using one of the many tools that exist for backing up MariaDB databases.

A simple approach that should work for the typically modestly sized seed database is `mysqldump`. The following commands should all be executed on the seed.

Backup

It should be safe to keep services running during the backup, but for maximum safety they may optionally be stopped:

```
docker exec -it bifrost_deploy \
systemctl stop ironic-api ironic-conductor ironic-inspector
```

Then, to perform the backup:

```
docker exec -it bifrost_deploy \
mysqldump --all-databases --single-transaction --routines --triggers >␣
↪seed-backup.sql
```

If the services were stopped prior to the backup, start them again:

```
docker exec -it bifrost_deploy \
systemctl start ironic-api ironic-conductor ironic-inspector
```

Restore

Prior to restoring the database, the Ironic and Ironic Inspector services should be stopped:

```
docker exec -it bifrost_deploy \
systemctl stop ironic-api ironic-conductor ironic-inspector
```

The database may then safely be restored:

```
docker exec -i bifrost_deploy \
mysql < seed-backup.sql
```

Finally, start the Ironic and Ironic Inspector services again:

```
docker exec -it bifrost_deploy \
systemctl start ironic-api ironic-conductor ironic-inspector
```

Running Commands

It is possible to run a command on the seed host:

```
(kayobe) $ kayobe seed host command run --command "<command>"
```

For example:

```
(kayobe) $ kayobe seed host command run --command "service docker restart"
```

Commands can also be run on the seed hypervisor host, if one is in use:

```
(kayobe) $ kayobe seed hypervisor host command run --command "<command>"
```

To execute the command with root privileges, add the `--become` argument. Adding the `--verbose` argument allows the output of the command to be seen.

3.9.3 Overcloud Administration

Updating Packages

It is possible to update packages on the overcloud hosts.

Package Repositories

If using custom package repositories, it may be necessary to update these prior to running a package update. To do this, update the configuration in `${KAYOBE_CONFIG_PATH}/dnf.yml` and run the following command:

```
(kayobe) $ kayobe overcloud host configure --tags dnf --kolla-tags none
```

Package Update

To update one or more packages:

```
(kayobe) $ kayobe overcloud host package update --packages <package1>,
↪<package2>
```

To update all eligible packages, use `*`, escaping if necessary:

```
(kayobe) $ kayobe overcloud host package update --packages "*"
```

To only install updates that have been marked security related:

```
(kayobe) $ kayobe overcloud host package update --packages "*" --security
```

Note that these commands do not affect packages installed in containers, only those installed on the host.

Kernel Updates

If the kernel has been updated, you will probably want to reboot the hosts to boot into the new kernel. This can be done using a command such as the following:

```
(kayobe) $ kayobe overcloud host command run --command "shutdown -r" --
↪become
```

It is normally best to apply this to control plane hosts in batches to avoid clustered services from losing quorum. This can be achieved using the `--limit` argument, and ensuring services are fully up after rebooting before proceeding with the next batch.

Running Commands

It is possible to run a command on the overcloud hosts:

```
(kayobe) $ kayobe overcloud host command run --command "<command>"
```

For example:

```
(kayobe) $ kayobe overcloud host command run --command "service docker␣
↪restart"
```

To execute the command with root privileges, add the `--become` argument. Adding the `--verbose` argument allows the output of the command to be seen.

Reconfiguring Containerised Services

When configuration is changed, it is necessary to apply these changes across the system in an automated manner. To reconfigure the overcloud, first make any changes required to the configuration on the Ansible control host. Next, run the following command:

```
(kayobe) $ kayobe overcloud service reconfigure
```

In case not all services configuration have been modified, performance can be improved by specifying Ansible tags to limit the tasks run in kayobe and/or kolla-ansibles playbooks. This may require knowledge of the inner workings of these tools but in general, kolla-ansible tags the play used to configure each service by the name of that service. For example: `nova`, `neutron` or `ironic`. Use `-t` or `--tags` to specify kayobe tags and `-kt` or `--kolla-tags` to specify kolla-ansible tags. For example:

```
(kayobe) $ kayobe overcloud service reconfigure --tags config --kolla-tags␣
↪nova,ironic
```

Deploying Updated Container Images

A common task is to deploy updated container images, without configuration changes. This might be to roll out an updated container OS or to pick up some package updates. This should be faster than a full deployment or reconfiguration.

To deploy updated container images:

```
(kayobe) $ kayobe overcloud service deploy containers
```

Note that if there are configuration changes, these will not be applied using this command so if in doubt, use a normal `kayobe overcloud service deploy`.

In case not all services containers have been modified, performance can be improved by specifying Ansible tags to limit the tasks run in kayobe and/or kolla-ansibles playbooks. This may require knowledge of the inner workings of these tools but in general, kolla-ansible tags the play used to configure each service by the name of that service. For example: `nova`, `neutron` or `ironic`. Use `-t` or `--tags` to specify kayobe tags and `-kt` or `--kolla-tags` to specify kolla-ansible tags. For example:

```
(kayobe) $ kayobe overcloud service deploy containers --kolla-tags nova,
↪ironic
```

Upgrading Containerised Services

Containerised control plane services may be upgraded by replacing existing containers with new containers using updated images which have been pulled from a registry or built locally. If using an updated version of Kayobe or upgrading from one release of OpenStack to another, be sure to follow the *kayobe upgrade guide*. It may be necessary to upgrade one or more services within a release, for example to apply a patch or minor release.

To upgrade the containerised control plane services:

```
(kayobe) $ kayobe overcloud service upgrade
```

As for the reconfiguration command, it is possible to specify tags for Kayobe and/or kolla-ansible:

```
(kayobe) $ kayobe overcloud service upgrade --tags config --kolla-tags␣
↪keystone
```

Stopping the Overcloud Services

Note: This step will stop all containers on the overcloud hosts.

To stop the overcloud services:

```
(kayobe) $ kayobe overcloud service stop --yes-i-really-really-mean-it
```

It should be noted that this state is persistent - containers will remain stopped after a reboot of the host on which they are running.

It is possible to limit the operation to particular hosts via `--kolla-limit`, or to particular services via `--kolla-tags`. It is also possible to avoid stopping the common containers via `--kolla-skip-tags common`. For example:

(kayobe) $ kayobe overcloud service stop kolla-tags glance,nova kolla-skip-tags common

Destroying the Overcloud Services

Note: This step will destroy all containers, container images, volumes and data on the overcloud hosts.

To destroy the overcloud services:

```
(kayobe) $ kayobe overcloud service destroy --yes-i-really-really-mean-it
```

Deprovisioning The Cloud

Note: This step will power down the overcloud hosts and delete their nodes instance state from the seeds ironic service.

To deprovision the overcloud:

```
(kayobe) $ kayobe overcloud deprovision
```

Saving Overcloud Service Configuration

It is often useful to be able to save the configuration of the control plane services for inspection or comparison with another configuration set prior to a reconfiguration or upgrade. This command will gather and save the control plane configuration for all hosts to the Ansible control host:

```
(kayobe) $ kayobe overcloud service configuration save
```

The default location for the saved configuration is `$PWD/overcloud-config`, but this can be changed via the `output-dir` argument. To gather configuration from a directory other than the default `/etc/kolla`, use the `node-config-dir` argument.

Generating Overcloud Service Configuration

Prior to deploying, reconfiguring, or upgrading a control plane, it may be useful to generate the configuration that will be applied, without actually applying it to the running containers. The configuration should typically be generated in a directory other than the default configuration directory of `/etc/kolla`, to avoid overwriting the active configuration:

```
(kayobe) $ kayobe overcloud service configuration generate --node-config-
↪dir /path/to/generated/config
```

The configuration will be generated remotely on the overcloud hosts in the specified directory, with one subdirectory per container. This command may be followed by `kayobe overcloud service configuration save` to gather the generated configuration to the Ansible control host.

Performing Database Backups

Database backups can be performed using the underlying support in Kolla Ansible.

In order to enable backups, enable Mariabackup in `${KAYOBE_CONFIG_PATH}/kolla.yml`:

```
kolla_enable_mariabackup: true
```

To apply this change, use the *kayobe overcloud service reconfigure* command.

To perform a full backup, run the following command:

```
kayobe overcloud database backup
```

Or to perform an incremental backup, run the following command:

```
kayobe overcloud database backup --incremental
```

Further information on backing up and restoring the database is available in the Kolla Ansible documentation.

Performing Database Recovery

Recover a completely stopped MariaDB cluster using the underlying support in Kolla Ansible.

To perform recovery run the following command:

```
kayobe overcloud database recover
```

Or to perform recovery on specified host, run the following command:

```
kayobe overcloud database recover --force-recovery-host <host>
```

By default the underlying kolla-ansible will automatically determine which host to use, and this option should not be used.

Gathering Facts

The following command may be used to gather facts for all overcloud hosts, for both Kayobe and Kolla Ansible:

```
kayobe overcloud facts gather
```

This may be useful to populate a fact cache in advance of other operations.

3.9.4 Baremetal Compute Node Management

When enrolling new hardware or performing maintenance, it can be useful to be able to manage many bare metal compute nodes simultaneously.

In all cases, commands are delegated to one of the controller hosts, and executed concurrently. Note that ansibles `forks` configuration option, which defaults to 5, may limit the number of nodes configured concurrently.

By default these commands wait for the state transition to complete for each node. This behavior can be changed by overriding the variable `baremetal_compute_wait` via `-e baremetal_compute_wait=False`

Manage

A node may need to be set to the `manageable` provision state in order to perform certain management operations, or when an enrolled node is transitioned into service. In order to manage a node, it must be in one of these states: `enroll`, `available`, `cleaning`, `clean failed`, `adopt failed` or `inspect failed`. To move the baremetal compute nodes to the `manageable` provision state:

```
(kayobe) $ kayobe baremetal compute manage
```

Provide

In order for nodes to be scheduled by nova, they must be `available`. To move the baremetal compute nodes from the `manageable` state to the `available` provision state:

```
(kayobe) $ kayobe baremetal compute provide
```

Inspect

Nodes must be in one of the following states: `manageable`, `inspect failed`, or `available`. To trigger hardware inspection on the baremetal compute nodes:

```
(kayobe) $ kayobe baremetal compute inspect
```

Rename

Once nodes have been discovered, it is helpful to associate them with a name to make them easier to work with. If you would like the nodes to be named according to their inventory host names, you can run the following command:

```
(kayobe) $ kayobe baremetal compute rename
```

This command will use the `ipmi_address` host variable from the inventory to map the inventory host name to the correct node.

Update Deployment Image

When the overcloud deployment images have been rebuilt or there has been a change to one of the following variables:

- `ipa_kernel_upstream_url`
- `ipa_ramdisk_upstream_url`

either by changing the url, or if the image to which they point has been changed, you need to update the `deploy_ramdisk` and `deploy_kernel` properties on the Ironic nodes. To do this you can run:

```
(kayobe) $ kayobe baremetal compute update deployment image
```

You can optionally limit the nodes in which this affects by setting `baremetal-compute-limit`:

```
(kayobe) $ kayobe baremetal compute update deployment image --baremetal-
↪compute-limit sand-6-1
```

which should take the form of an ansible host pattern. This is matched against the Ironic node name.

Ironic Serial Console

To access the baremetal nodes from within Horizon you need to enable the serial console. For this to work the you must set `kolla_enable_nova_serialconsole_proxy` to `true` in `etc/kayobe/kolla.yml`:

```
kolla_enable_nova_serialconsole_proxy: true
```

The console interface on the Ironic nodes is expected to be `ipmitool-socat`, you can check this with:

```
openstack baremetal node show <node_id> --fields console_interface
```

where <node_id> should be the UUID or name of the Ironic node you want to check.

If you have set `kolla_ironic_enabled_console_interfaces` in `etc/kayobe/ironic.yml`, it should include `ipmitool-socat` in the list of enabled interfaces.

The playbook to enable the serial console currently only works if the Ironic node name matches the inventory hostname.

Once these requirements have been satisfied, you can run:

```
(kayobe) $ kayobe baremetal compute serial console enable
```

This will reserve a TCP port for each node to use for the serial console interface. The allocations are stored in `${KAYOBE_CONFIG_PATH}/console-allocation.yml`. The current implementation uses a global pool, which is specified by `ironic_serial_console_tcp_pool_start` and `ironic_serial_console_tcp_pool_end`; these variables can set in `etc/kayobe/ironic.yml`.

To disable the serial console you can use:

```
(kayobe) $ kayobe baremetal compute serial console disable
```

The port allocated for each node is retained and must be manually removed from `${KAYOBE_CONFIG_PATH}/console-allocation.yml` if you want it to be reused by another Ironic node with a different name.

You can optionally limit the nodes targeted by setting `baremetal-compute-limit`:

```
(kayobe) $ kayobe baremetal compute serial console enable --baremetal-
↪compute-limit sand-6-1
```

which should take the form of an ansible host pattern.

Serial console auto-enable

To enable the serial consoles automatically on `kayobe overcloud post configure`, you can set `ironic_serial_console_autoenable` in `etc/kayobe/ironic.yml`:

```
ironic_serial_console_autoenable: true
```

3.10 Resources

This section contains links to external Kayobe resources.

3.10.1 A Universe From Nothing

Note: The A Universe From Nothing deployment guide is intended for educational & testing purposes only. It is *not* production ready.

Originally created as a workshop, A Universe From Nothing is an example guide for the deployment of Kayobe on virtual hardware. You can find it on GitHub here.

The repository contains a configuration suitable for deploying containerised OpenStack using Kolla, Ansible and Kayobe. The guide makes use of Tenks to provision a virtual baremetal environment running on a single hypervisor.

To complete the walkthrough you will require a baremetal or VM hypervisor running CentOS 8 with at least 32GB RAM & 80GB disk space. Preparing the deployment can take some time - where possible it is beneficial to snapshot the hypervisor. We advise making a snapshot after creating the initial seed VM as this will make additional deployments significantly faster.

3.11 Advanced Documentation

3.11.1 Control Plane Service Placement

Note: This is an advanced topic and should only be attempted when familiar with kayobe and OpenStack.

The default configuration in kayobe places all control plane services on a single set of servers described as controllers. In some cases it may be necessary to introduce more than one server role into the control plane, and control which services are placed onto the different server roles.

Configuration

Overcloud Inventory Discovery

If using a seed host to enable discovery of the control plane services, it is necessary to configure how the discovered hosts map into kayobe groups. This is done using the `overcloud_group_hosts_map` variable, which maps names of kayobe groups to a list of the hosts to be added to that group.

This variable will be used during the command `kayobe overcloud inventory discover`. An inventory file will be generated in `${KAYOBE_CONFIG_PATH}/inventory/overcloud` with discovered hosts added to appropriate kayobe groups based on `overcloud_group_hosts_map`.

Kolla-ansible Inventory Mapping

Once hosts have been discovered and enrolled into the kayobe inventory, they must be added to the kolla-ansible inventory. This is done by mapping from top level kayobe groups to top level kolla-ansible groups using the `kolla_overcloud_inventory_top_level_group_map` variable. This variable maps from kolla-ansible groups to lists of kayobe groups, and variables to define for those groups in the kolla-ansible inventory.

Variables For Custom Server Roles

Certain variables must be defined for hosts in the `overcloud` group. For hosts in the `controllers` group, many variables are mapped to other variables with a `controller_` prefix in files under `ansible/group_vars/controllers/`. This is done in order that they may be set in a global extra variables file, typically `controllers.yml`, with defaults set in `ansible/group_vars/all/controllers`. A similar scheme is used for hosts in the `monitoring` group.

Table 2: Overcloud host variables

Variable	Purpose
`ansible_user`	Username with which to access the host via SSH.
`bootstrap_user`	Username with which to access the host before `ansible_user` is configured.
`lvm_groups`	List of LVM volume groups to configure. See mrlesmithjr.manage-lvm role for format.
`mdadm_arrays`	List of software RAID arrays. See mrlesmithjr.mdadm role for format.
`network_interfaces`	List of names of networks to which the host is connected.
`sysctl_parameters`	Dict of sysctl parameters to set.
`users`	List of users to create. See singleplatform-eng.users role

If configuring BIOS and RAID via `kayobe overcloud bios raid configure`, the following variables should also be defined:

Table 3: Overcloud BIOS & RAID host variables

Variable	Purpose
`bios_config`	Dict mapping BIOS configuration options to their required values. See stackhpc.drac role for format.
`raid_config`	List of RAID virtual disks to configure. See stackhpc.drac role for format.

These variables can be defined in inventory host or group variables files, under `${KAYOBE_CONFIG_PATH}/inventory/host_vars/<host>` or `${KAYOBE_CONFIG_PATH}/inventory/group_vars/<group>` respectively.

Custom Kolla-ansible Inventories

As an advanced option, it is possible to fully customise the content of the kolla-ansible inventory, at various levels. To facilitate this, kayobe breaks the kolla-ansible inventory into three separate sections.

Top level groups define the roles of hosts, e.g. `controller` or `compute`, and it is to these groups that hosts are mapped directly.

Components define groups of services, e.g. `nova` or `ironic`, which are mapped to top level groups.

Services define single containers, e.g. `nova-compute` or `ironic-api`, which are mapped to components.

The default top level inventory is generated from `kolla_overcloud_inventory_top_level_group_map`. Kayobes component- and service-level inventory for kolla-ansible is static, and taken from the kolla-ansible example `multinode` inventory. The complete inventory is generated by concatenating these inventories.

Each level may be separately overridden by setting the following variables:

Table 4: Custom kolla-ansible inventory variables

Variable	Purpose
`kolla_overcloud_inventory_custom_top_level`	Overcloud inventory containing a mapping from top level groups to hosts.
`kolla_overcloud_inventory_custom_components`	Overcloud inventory containing a mapping from components to top level groups.
`kolla_overcloud_inventory_custom_services`	Overcloud inventory containing a mapping from services to components.
`kolla_overcloud_inventory_custom`	Full overcloud inventory contents.

Examples

Example 1: Adding Network Hosts

This example walks through the configuration that could be applied to enable the use of separate hosts for neutron network services and load balancing. The control plane consists of three controllers, `controller-[0-2]`, and two network hosts, `network-[0-1]`. All file paths are relative to `${KAYOBE_CONFIG_PATH}`.

First, we must make the network group separate from controllers:

Listing 121: `inventory/groups`

```
[controllers]
# Empty group to provide declaration of controllers group.

[network]
# Empty group to provide declaration of network group.
```

Then, we must map the hosts to kayobe groups.

Listing 122: `overcloud.yml`

```
overcloud_group_hosts_map:
  controllers:
    - controller-0
    - controller-1
    - controller-2
  network:
    - network-0
    - network-1
```

Next, we must map these groups to kolla-ansible groups.

Listing 123: `kolla.yml`

```
kolla_overcloud_inventory_top_level_group_map:
  control:
    groups:
      - controllers
  network:
    groups:
      - network
```

Finally, we create a group variables file for hosts in the network group, providing the necessary variables for a control plane host.

Listing 124: `inventory/group_vars/network`

```
ansible_user: "{{ kayobe_ansible_user }}"
bootstrap_user: "{{ controller_bootstrap_user }}"
lvm_groups: "{{ controller_lvm_groups }}"
mdadm_arrays: "{{ controller_mdadm_arrays }}"
network_interfaces: "{{ controller_network_host_network_interfaces }}"
sysctl_parameters: "{{ controller_sysctl_parameters }}"
users: "{{ controller_users }}"
```

Here we are using the controller-specific values for some of these variables, but they could equally be different.

Example 2: Overriding the Kolla-ansible Inventory

This example shows how to override one or more sections of the kolla-ansible inventory. All file paths are relative to `${KAYOBE_CONFIG_PATH}`.

It is typically best to start with an inventory template taken from the Kayobe source code, and then customize it. The templates can be found in `ansible/roles/kolla-ansible/templates`, e.g. components template is `overcloud-components.j2`.

First, create a file containing the customised inventory section. Well use the **components** section in this example.

Listing 125: `kolla/inventory/overcloud-components.j2`

```
[nova]
control

[ironic]
{% if kolla_enable_ironic | bool %}
control
{% endif %}

...
```

Next, we must configure kayobe to use this inventory template.

Listing 126: `kolla.yml`

```
kolla_overcloud_inventory_custom_components: "{{ lookup('template', kayobe_
↪config_path ~ '/kolla/inventory/overcloud-components.j2') }}"
```

Here we use the `template` lookup plugin to render the Jinja2-formatted inventory template.

3.11.2 Custom Ansible Playbooks

Kayobe supports running custom Ansible playbooks located outside of the kayobe project. This provides a flexible mechanism for customising a control plane. Access to the kayobe variables is possible, ensuring configuration does not need to be repeated.

Kayobe Custom Playbook API

Explicitly allowing users to run custom playbooks with access to the kayobe variables elevates the variable namespace and inventory to become an interface. This raises questions about the stability of this interface, and the guarantees it provides.

The following guidelines apply to the custom playbook API:

- Only variables defined in the kayobe configuration files under `etc/kayobe` are supported.
- The groups defined in `etc/kayobe/inventory/groups` are supported.
- Any change to a supported variable (rename, schema change, default value change, or removal) or supported group (rename or removal) will follow a deprecation period of one release cycle.
- Kayobes internal roles may not be used.

Note that these are guidelines, and exceptions may be made where appropriate.

Running Custom Ansible Playbooks

Run one or more custom ansible playbooks:

```
(kayobe) $ kayobe playbook run <playbook>[, <playbook>...]
```

Playbooks do not by default have access to the Kayobe playbook group variables, filter plugins, and test plugins, since these are relative to the current playbooks directory. This can be worked around by creating symbolic links to the Kayobe repository from the Kayobe configuration.

Packaging Custom Playbooks With Configuration

The kayobe project encourages its users to manage configuration for a cloud using version control, based on the kayobe-config repository. Storing custom Ansible playbooks in this repository makes a lot of sense, and kayobe has special support for this.

It is recommended to store custom playbooks in `$KAYOBE_CONFIG_PATH/ansible/`. Roles located in `$KAYOBE_CONFIG_PATH/ansible/roles/` will be automatically available to playbooks in this directory.

With this directory layout, the following commands could be used to create symlinks that allow access to Kayobes filter plugins, group variables and test plugins:

```
cd ${KAYOBE_CONFIG_PATH}/ansible/
ln -s ../../../../kayobe/ansible/filter_plugins/ filter_plugins
ln -s ../../../../kayobe/ansible/group_vars/ group_vars
ln -s ../../../../kayobe/ansible/test_plugins/ test_plugins
```

These symlinks can even be committed to the kayobe-config Git repository.

Note: These symlinks rely on having a kayobe source checkout at the same level as the kayobe-config repository checkout, as described in *Installation from source*.

Ansible Galaxy

Ansible Galaxy provides a means for sharing Ansible roles. Kayobe configuration may provide a Galaxy requirements file that defines roles to be installed from Galaxy. These roles may then be used by custom playbooks.

Galaxy role dependencies may be defined in `$KAYOBE_CONFIG_PATH/ansible/requirements.yml`. These roles will be installed in `$KAYOBE_CONFIG_PATH/ansible/roles/` when bootstrapping the Ansible control host:

```
(kayobe) $ kayobe control host bootstrap
```

And updated when upgrading the Ansible control host:

```
(kayobe) $ kayobe control host upgrade
```

Example

The following example adds a `foo.yml` playbook to a set of kayobe configuration. The playbook uses a Galaxy role, `bar.baz`.

Here is the kayobe configuration repository structure:

```
etc/kayobe/
    ansible/
        foo.yml
        requirements.yml
        roles/
    bifrost.yml
...
```

Here is the playbook, `ansible/foo.yml`:

```
---
- hosts: controllers
  roles:
    - name: bar.baz
```

Here is the Galaxy requirements file, `ansible/requirements.yml`:

```
---
- bar.baz
```

We should first install the Galaxy role dependencies, to download the `bar.baz` role:

```
(kayobe) $ kayobe control host bootstrap
```

Then, to run the `foo.yml` playbook:

```
(kayobe) $ kayobe playbook run $KAYOBE_CONFIG_PATH/ansible/foo.yml
```

Hooks

Warning: Hooks are an experimental feature and the design could change in the future. You may have to update your config if there are any changes to the design. This warning will be removed when the design has been stabilised.

Hooks allow you to automatically execute custom playbooks at certain points during the execution of a kayobe command. The point at which a hook is run is referred to as a `target`. Please see the *list of available targets*.

Hooks are created by symlinking an existing playbook into the the relevant directory under `$KAYOBE_CONFIG_PATH/hooks`. Kayobe will search the hooks directory for sub-directories matching `<command>.<target>.d`, where `command` is the name of a kayobe command with any spaces replaced with dashes, and `target` is one of the supported targets for the command.

For example, when using the command:

```
(kayobe) $ kayobe control host bootstrap
```

kayobe will search the paths:

- `$KAYOBE_CONFIG_PATH/hooks/control-host-bootstrap/pre.d`
- `$KAYOBE_CONFIG_PATH/hooks/control-host-bootstrap/post.d`

Any playbooks listed under the `pre.d` directory will be run before kayobe executes its own playbooks and any playbooks under `post.d` will be run after. You can affect the order of the playbooks by prefixing the symlink with a sequence number. The sequence number must be separated from the hook name with a dash. Playbooks with smaller sequence numbers are run before playbooks with larger ones. Any ties are broken by alphabetical ordering.

For example to run the playbook `foo.yml` after `kayobe overcloud host configure`, you could do the following:

```
(kayobe) $ mkdir -p ${KAYOBE_CONFIG_PATH}/hooks/overcloud-host-configure/
↪post.d
(kayobe) $ cd ${KAYOBE_CONFIG_PATH}/hooks/overcloud-host-configure/post.d
(kayobe) $ ln -s ../../../ansible/foo.yml 10-foo.yml
```

The sequence number for the `foo.yml` playbook is `10`.

Failure handling

If the exit status of any playbook, including built-in playbooks and custom hooks, is non-zero, kayobe will not run any subsequent hooks or built-in kayobe playbooks. Ansible provides several methods for preventing a task from producing a failure. Please see the Ansible documentation for more details. Below is an example showing how you can use the `ignore_errors` option to prevent a task from causing the playbook to report a failure:

```
---
- name: Failure example
  hosts: localhost
  tasks:
    - name: Deliberately fail
      fail:
      ignore_errors: true
```

A failure in the `Deliberately fail` task would not prevent subsequent tasks, hooks, and playbooks from running.

Targets

The following targets are available for all commands:

Table 5: all commands

Target	Description
pre	Runs before a kayobe command has start executing
post	Runs after a kayobe command has finished executing

3.12 Contributor Guide

3.12.1 Contributor Guide

This guide is for contributors of the Kayobe project. It includes information on proposing your first patch and how to participate in the community. It also covers responsibilities of core reviewers and the Project Team Lead (PTL), and information about development processes.

We welcome everyone to join our project!

So You Want to Contribute

For general information on contributing to OpenStack, please check out the contributor guide to get started. It covers all the basics that are common to all OpenStack projects: the accounts you need, the basics of interacting with our Gerrit review system, how we communicate as a community, etc.

Below will cover the more project specific information you need to get started with Kayobe.

Basics

The source repository for this project can be found at:

https://opendev.org/openstack/kayobe

Communication

Kayobe shares communication channels with Kolla.

IRC Channel `#openstack-kolla` (channel logs) on OFTC

Weekly Meetings On Wednesdays at 15:00 UTC in the IRC channel (meetings logs)

Mailing list (prefix subjects with `[kolla]`) http://lists.openstack.org/pipermail/openstack-discuss/

Meeting Agenda https://wiki.openstack.org/wiki/Meetings/Kolla

Whiteboard (etherpad) Keeping track of CI gate status, release status, stable backports, planning and feature development status. https://etherpad.openstack.org/p/KollaWhiteBoard

Contacting the Core Team

The list in alphabetical order (on first name):

Name	IRC nick	Email address
Doug Szumski	dougsz	doug@stackhpc.com
John Garbutt	johnthetubaguy	john@johngarbutt.com
Kevin Tibi	ktibi	kevintibi@hotmail.com
Mark Goddard	mgoddard	mark@stackhpc.com
Pierre Riteau	priteau	pierre@stackhpc.com
Will Szumski	jovial	will@stackhpc.com

The current effective list is also available from Gerrit: https://review.opendev.org/#/admin/groups/1875,members

New Feature Planning

New features are discussed via IRC or mailing list (with [kolla] prefix). Kayobe project keeps RFEs in Storyboard. Specs are welcome but not strictly required.

Task Tracking

Kolla project tracks tasks in Storyboard. Note this is the same place as for bugs.

A more lightweight task tracking is done via etherpad - Whiteboard.

Reporting a Bug

You found an issue and want to make sure we are aware of it? You can do so on Storyboard. Note this is the same place as for tasks.

Getting Your Patch Merged

Most changes proposed to Kayobe require two +2 votes from core reviewers before +W. A release note is required on most changes as well. Release notes policy is described in *its own section*.

Significant changes should have documentation and testing provided with them.

Project Team Lead Duties

All common PTL duties are enumerated in the PTL guide. Release tasks are described in the *Kayobe releases guide*.

Development

Source Code Orientation

There are a number of layers to Kayobe, so here we provide a few pointers to the major parts.

CLI

The Command Line Interface (CLI) is built using the cliff library. Commands are exposed as Python entry points in setup.cfg. These entry points map to classes in kayobe/cli/commands.py. The helper modules kayobe/ansible.py and kayobe/kolla_ansible.py are used to execute Kayobe playbooks and Kolla Ansible commands respectively.

Ansible

Kayobes Ansible playbooks live in ansible/*.yml, and these typically execute roles in ansible/roles/. Global variable defaults are defined in group variable files in ansible/group_vars/all/ and these typically map to commented out variables in the configuration files in etc/kayobe/*.yml. A number of custom Jinja filters exist in ansible/filter_plugins/*.py. Kayobe depends on roles hosted on Ansible Galaxy, and these and their version requirements are defined in requirements.yml.

Ansible Galaxy

Kayobe uses a number of Ansible roles hosted on Ansible Galaxy. The role dependencies are tracked in `requirements.yml`, and specify required versions. The process for changing a Galaxy role is as follows:

1. If required, develop changes for the role. This may be done outside of Kayobe, or by modifying the role in place during development. If upstream changes to the role have already been made, this step can be skipped.
2. Commit changes to the role, typically via a Github pull request.
3. Request that a tagged release of the role be made, or make one if you have the necessary privileges.
4. Ensure that automatic imports are configured for the role using e.g. a TravisCI webhook notification, or perform a manual import of the role on Ansible Galaxy.
5. Modify the version in `requirements.yml` to match the new release of the role.

Vagrant

Kayobe provides a Vagrantfile that can be used to bring up a virtual machine for use as a development environment. The VM is based on the centos/8 CentOS 8 image, and supports the following providers:

- VirtualBox
- VMWare Fusion

The VM is configured with 4GB RAM and a 20GB HDD. It has a single private network in addition to the standard Vagrant NAT network.

Preparation

First, ensure that Vagrant is installed and correctly configured to use the required provider. Also install the following vagrant plugins:

```
vagrant plugin install vagrant-reload vagrant-disksize
```

If using the VirtualBox provider, install the following vagrant plugin:

```
vagrant plugin install vagrant-vbguest
```

Note: if using Ubuntu 16.04 LTS, you may be unable to install any plugins. To work around this install the upstream version from www.virtualbox.org.

Usage

Later sections in the development guide cover in more detail how to use the development VM in different configurations. These steps cover bringing up and accessing the VM.

Clone the kayobe repository:

```
git clone https://opendev.org/openstack/kayobe.git -b stable/
↪victoria
```

Change the current directory to the kayobe repository:

```
cd kayobe
```

Inspect kayobes `Vagrantfile`, noting the provisioning steps:

```
less Vagrantfile
```

Bring up a virtual machine:

```
vagrant up
```

Wait for the VM to boot, then SSH in:

```
vagrant ssh
```

Manual Setup

This section provides a set of manual steps to set up a development environment for an OpenStack controller in a virtual machine using Vagrant and Kayobe.

For a more automated and flexible procedure, see *Automated Setup*.

Preparation

Follow the steps in *Vagrant* to prepare your environment for use with Vagrant and bring up a Vagrant VM.

Manual Installation

Sometimes the best way to learn a tool is to ditch the scripts and perform a manual installation.

SSH into the controller VM:

```
vagrant ssh
```

Source the kayobe virtualenv activation script:

```
source kayobe-venv/bin/activate
```

Change the current directory to the Vagrant shared directory:

```
cd /vagrant
```

Source the kayobe environment file:

```
source kayobe-env
```

Bootstrap the kayobe Ansible control host:

```
kayobe control host bootstrap
```

Configure the controller host:

```
kayobe overcloud host configure
```

At this point, container images must be acquired. They can either be built locally or pulled from an image repository if appropriate images are available.

Either build container images:

```
kayobe overcloud container image build
```

Or pull container images:

```
kayobe overcloud container image pull
```

Deploy the control plane services:

```
kayobe overcloud service deploy
```

Source the OpenStack environment file:

```
source ${KOLLA_CONFIG_PATH:-/etc/kolla}/admin-openrc.sh
```

Perform post-deployment configuration:

```
kayobe overcloud post configure
```

Next Steps

The OpenStack control plane should now be active. Try out the following:

- register a user
- create an image
- upload an SSH keypair
- access the horizon dashboard

The cloud is your oyster!

To Do

Create virtual baremetal nodes to be managed by the OpenStack control plane.

Automated Setup

This section provides information on the development tools provided by Kayobe to automate the deployment of various development environments.

For a manual procedure, see *Manual Setup*.

Overview

The Kayobe development environment automation tooling is built using simple shell scripts. Some minimal configuration can be applied by setting the environment variables in *dev/config.sh*. Control plane configuration is typically provided via the kayobe-config-dev repository, although it is also possible to use your own Kayobe configuration. This allows us to build a development environment that is as close to production as possible.

Environments

The following development environments are supported:

- Overcloud (single OpenStack controller)
- Seed

The Universe from Nothing workshop may be of use for more advanced testing scenarios involving a seed hypervisor, seed VM, and separate control and compute nodes.

Overcloud

Preparation

Clone the Kayobe repository:

```
git clone https://opendev.org/openstack/kayobe.git -b stable/
↪victoria
```

Change the current directory to the Kayobe repository:

```
cd kayobe
```

Clone the `kayobe-config-dev` repository to `config/src/kayobe-config`

```
mkdir -p config/src
git clone https://opendev.org/openstack/kayobe-config-dev.git␣
↪config/src/kayobe-config -b stable/victoria
```

Inspect the Kayobe configuration and make any changes necessary for your environment.

If using Vagrant, follow the steps in *Vagrant* to prepare your environment for use with Vagrant and bring up a Vagrant VM.

If not using Vagrant, the default development configuration expects the presence of a bridge interface on the OpenStack controller host to carry control plane traffic. The bridge should be named `breth1` with a single port `eth1`, and an IP address of `192.168.33.3/24`. This can be modified by editing `config/src/kayobe-config/etc/kayobe/inventory/group_vars/controllers/network-interfaces`. Alternatively, this can be added using the following commands:

```
sudo ip l add breth1 type bridge
sudo ip l set breth1 up
sudo ip a add 192.168.33.3/24 dev breth1
sudo ip l add eth1 type dummy
sudo ip l set eth1 up
sudo ip l set eth1 master breth1
```

Usage

If using Vagrant, SSH into the Vagrant VM and change to the shared directory:

```
vagrant ssh
cd /vagrant
```

If not using Vagrant, run the `dev/install-dev.sh` script to install Kayobe and its dependencies in a Python virtual environment:

```
./dev/install-dev.sh
```

Note: This will create an *editable install*. It is also possible to install Kayobe in a non-editable way, such that changes will not been seen until you reinstall the package. To do this you can run `./dev/install.sh`.

Run the `dev/overcloud-deploy.sh` script to deploy the OpenStack control plane:

```
./dev/overcloud-deploy.sh
```

Upon successful completion of this script, the control plane will be active.

Testing

Scripts are provided for testing the creation of virtual and bare metal instances.

Virtual Machines

The control plane can be tested by running the `dev/overcloud-test-vm.sh` script. This will run the `init-runonce` setup script provided by Kolla Ansible that registers images, networks, flavors etc. It will then deploy a virtual server instance, and delete it once it becomes active:

```
./dev/overcloud-test-vm.sh
```

Bare Metal Compute

For a control plane with Ironic enabled, a bare metal instance can be deployed. We can use the Tenks project to create fake bare metal nodes.

Clone the tenks repository:

```
git clone https://opendev.org/openstack/tenks.git
```

Optionally, edit the Tenks configuration file, `dev/tenks-deploy-config-compute.yml`.

Run the `dev/tenks-deploy-compute.sh` script to deploy Tenks:

```
./dev/tenks-deploy-compute.sh ./tenks
```

Check that Tenks has created VMs called `tk0` and `tk1`:

```
sudo virsh -c qemu+unix:///system?socket=/var/run/libvirt-tenks/libvirt-
↪sock list --all
```

Verify that VirtualBMC is running:

```
~/tenks-venv/bin/vbmc list
```

Configure the firewall to allow the baremetal nodes to access OpenStack services:

```
./dev/configure-firewall.sh
```

We are now ready to run the `dev/overcloud-test-baremetal.sh` script. This will run the `init-runonce` setup script provided by Kolla Ansible that registers images, networks, flavors etc. It will then deploy a bare metal server instance, and delete it once it becomes active:

```
./dev/overcloud-test-baremetal.sh
```

The machines and networking created by Tenks can be cleaned up via `dev/tenks-teardown-compute.sh`:

```
./dev/tenks-teardown-compute.sh ./tenks
```

Upgrading

It is possible to test an upgrade from a previous release by running the `dev/overcloud-upgrade.sh` script:

```
./dev/overcloud-upgrade.sh
```

Seed

These instructions cover deploying the seed services directly rather than in a VM.

Preparation

Clone the Kayobe repository:

```
git clone https://opendev.org/openstack/kayobe.git -b stable/
↪victoria
```

Change to the `kayobe` directory:

```
cd kayobe
```

Clone the `kayobe-config-dev` repository to `config/src/kayobe-config`:

```
mkdir -p config/src
git clone https://opendev.org/openstack/kayobe-config-dev.git␣
↪config/src/kayobe-config -b stable/victoria
```

Inspect the Kayobe configuration and make any changes necessary for your environment.

The default development configuration expects the presence of a bridge interface on the seed host to carry provisioning traffic. The bridge should be named `breth1` with a single port `eth1`, and an IP address of `192.168.33.5/24`. This can be modified by editing `config/src/kayobe-config/etc/kayobe/inventory/group_vars/seed/network-interfaces`. Alternatively, this can be added using the following commands:

```
sudo ip l add breth1 type bridge
sudo ip l set breth1 up
sudo ip a add 192.168.33.5/24 dev breth1
sudo ip l add eth1 type dummy
sudo ip l set eth1 up
sudo ip l set eth1 master breth1
```

Usage

Run the `dev/install.sh` script to install Kayobe and its dependencies in a Python virtual environment:

```
./dev/install.sh
```

Run the `dev/seed-deploy.sh` script to deploy the seed services:

```
export KAYOBE_SEED_VM_PROVISION=0
./dev/seed-deploy.sh
```

Upon successful completion of this script, the seed will be active.

Testing

The seed services may be tested using the Tenks project to create fake bare metal nodes.

If your seed has a non-standard MTU, you should set it via `aio_mtu` in `etc/kayobe/networks.yml`.

Clone the tenks repository:

```
git clone https://opendev.org/openstack/tenks.git
```

Optionally, edit the Tenks configuration file, `dev/tenks-deploy-config-overcloud.yml`.

Run the `dev/tenks-deploy-overcloud.sh` script to deploy Tenks:

```
./dev/tenks-deploy-overcloud.sh ./tenks
```

Check that Tenks has created a VM called `controller0`:

```
sudo virsh list --all
```

Verify that VirtualBMC is running:

```
~/tenks-venv/bin/vbmc list
```

It is now possible to discover, inspect and provision the controller VM:

```
source dev/environment-setup.sh
kayobe overcloud inventory discover
kayobe overcloud hardware inspect
kayobe overcloud provision
```

The controller VM is now accessible via SSH as the bootstrap user (`centos` or `ubuntu`) at `192.168.33.3`.

The machines and networking created by Tenks can be cleaned up via `dev/tenks-teardown-overcloud.sh`:

```
./dev/tenks-teardown-overcloud.sh ./tenks
```

Upgrading

It is possible to test an upgrade by running the `dev/seed-upgrade.sh` script:

```
./dev/seed-upgrade.sh
```

Testing

Kayobe has a number of test suites covering different areas of code. Many tests are run in virtual environments using `tox`.

Preparation

System Prerequisites

The following packages should be installed on the development system prior to running kayobes tests.

- Ubuntu/Debian:

```
sudo apt-get install build-essential python3-dev libssl-dev python3-
↪pip git
```

- Fedora or CentOS/RHEL 8:

```
sudo dnf install python3-devel openssl-devel python3-pip git gcc
```

- OpenSUSE/SLE 12:

```
sudo zypper install python3-devel python3-pip libopenssl-devel git
```

Python Prerequisites

If your distro has at least `tox 1.8`, use your system package manager to install the `python-tox` package. Otherwise install this on all distros:

```
sudo pip install -U tox
```

You may need to explicitly upgrade `virtualenv` if youve installed the one from your OS distribution and it is too old (tox will complain). You can upgrade it individually, if you need to:

```
sudo pip install -U virtualenv
```

Running Unit Tests Locally

If you havent already, the kayobe source code should be pulled directly from git:

```
# from your home or source directory
cd ~
git clone https://opendev.org/openstack/kayobe.git -b stable/
↪victoria
cd kayobe
```

Running Unit and Style Tests

Kayobe defines a number of different tox environments in `tox.ini`. The default environments may be displayed:

```
tox -list
```

To run all default environments:

```
tox
```

To run one or more specific environments, including any of the non-default environments:

```
tox -e <environment>[,<environment>]
```

Environments

The following tox environments are provided:

alint Run Ansible linter.

ansible Run Ansible tests for some ansible roles using Ansible playbooks.

ansible-syntax Run a syntax check for all Ansible files.

docs Build Sphinx documentation.

molecule Run Ansible tests for some Ansible roles using the molecule test framework.

pep8 Run style checks for all shell, python and documentation files.

py3 Run python unit tests for kayobe python module.

Writing Tests

Unit Tests

Unit tests follow the lead of OpenStack, and use `unittest`. One difference is that tests are run using the discovery functionality built into `unittest`, rather than `ostestr`/`stestr`. Unit tests are found in `kayobe/tests/unit/`, and should be added to cover all new python code.

Ansible Role Tests

Two types of test exist for Ansible roles - pure Ansible and molecule tests.

Pure Ansible Role Tests

These tests exist for the `kolla-ansible` role, and are found in `ansible/<role>/tests/*.yml`. The role is exercised using an ansible playbook.

Molecule Role Tests

Molecule is an Ansible role testing framework that allows roles to be tested in isolation, in a stable environment, under multiple scenarios. Kayobe uses Docker engine to provide the test environment, so this must be installed and running on the development system.

Molecule scenarios are found in `ansible/<role>/molecule/<scenario>`, and defined by the config file `ansible/<role>/molecule/<scenario>/molecule.yml` Tests are written in python using the pytest framework, and are found in `ansible/<role>/molecule/<scenario>/tests/test_*.py`.

Molecule tests currently exist for the `kolla-openstack` role, and should be added for all new roles where practical.

Release notes

Kayobe (just like Kolla) uses the following release notes sections:

- `features` for new features or functionality; these should ideally refer to the blueprint being implemented;
- `fixes` for fixes closing bugs; these must refer to the bug being closed;
- `upgrade` for notes relevant when upgrading from previous version; these should ideally be added only between major versions; required when the proposed change affects behaviour in a non-backwards compatible way or generally changes something impactful;
- `deprecations` to track deprecated features; relevant changes may consist of only the commit message and the release note;
- `prelude` filled in by the PTL before each release or RC.

Other release note types may be applied per common sense. Each change should include a release note unless being a `TrivialFix` change or affecting only docs or CI. Such changes should *not* include a release note to avoid confusion. Remember release notes are mostly for end users which, in case of Kolla, are OpenStack administrators/operators. In case of doubt, the core team will let you know what is required.

To add a release note, run the following command:

```
tox -e venv -- reno new <summary-line-with-dashes>
```

All release notes can be inspected by browsing `releasenotes/notes` directory.

To generate release notes in HTML format in `releasenotes/build`, run:

```
tox -e releasenotes
```

Note this requires the release note to be tracked by `git` so you have to at least add it to the `git`s staging area.

Releases

This guide is intended to complement the OpenStack releases site, and the project team guides section on release management.

Team members make themselves familiar with the release schedule for the current release, for example https://releases.openstack.org/train/schedule.html.

Release Model

As a deployment project, Kayobes release model differs from many other OpenStack projects. Kayobe follows the cycle-trailing release model, to allow time after the OpenStack coordinated release to wait for distribution packages and support new features. This gives us three months after the final release to prepare our final releases. Users are typically keen to try out the new release, so we should aim to release as early as possible while ensuring we have confidence in the release.

Release Schedule

While we dont wish to repeat the OpenStack release documentation, we will point out the high level schedule, and draw attention to areas where our process is different.

Milestones

At each of the various release milestones, pay attention to what other projects are doing.

Feature Freeze

As with projects following the common release model, Kayobe uses a feature freeze period to allow the code to stabilise prior to release. There is no official feature freeze date for the cycle-trailing model, but we typically freeze around **three weeks** after the common feature freeze. During this time, no features should be merged to the master branch.

Before RC1

Prior to creating a release candidate and stable branch, the following tasks should be performed.

Testing

Test the code and fix at a minimum all critical issues.

Update dependencies to upcoming release

Prior to the release, we update the dependencies and upper constraints on the master branch to use the upcoming release. This is now quite easy to do, following the introduction of the `openstack_release` variable. This is done prior to creating a release candidate. For example, see https://review.opendev.org/#/c/694616/.

Synchronise kayobe-config

Ensure that configuration defaults in `kayobe-config` are in sync with those under `etc/kayobe` in `kayobe`. This can be done via:

```
cp -aR kayobe/etc/kayobe/* kayobe-config/etc/kayobe
```

Commit the changes and submit for review.

Synchronise kayobe-config-dev

Ensure that configuration defaults in `kayobe-config-dev` are in sync with those in `kayobe-config`. This requires a little more care, since some configuration options have been changed from the defaults. Choose a method to suit you and be careful not to lose any configuration.

Commit the changes and submit for review.

Prepare release notes

Its possible to add a prelude to the release notes for a particular release using a `prelude` section in a `reno` note.

Ensure that release notes added during the release cycle are tidy and consistent. The following command is useful to list release notes added this cycle:

```
git diff --name-only origin/stable/<previous release> -- releasenotes/
```

RC1

Prior to cutting a stable branch, the `master` branch should be tagged as a release candidate. This allows the `reno` tool to determine where to stop searching for release notes for the next release. The tag should take the following form: `<release tag>.0rc$n`, where `$n` is the release candidate number.

This should be done for each deliverable using the releases tooling. A release candidate and stable branch defintitions should be added for each Kayobe deliverable (`kayobe`, `kayobe-config`, `kayobe-config-dev`). These are defined in `deliverables/<release name>/kayobe.yaml`. Currently the same version is used for each deliverable.

The changes should be proposed to the releases repository. For example: https://review.opendev.org/#/c/700174.

After RC1

The OpenStack proposal bot will propose changes to the new branch and the master branch. These need to be approved.

After the stable branch has been cut, the master branch can be unfrozen and development on features for the next release can begin. At this point it will still be using dependencies and upper constraints from the release branch, so revert the patch created in *Update dependencies to upcoming release*. For example, see https://review.opendev.org/701747.

Finally, set the previous release used in upgrade jobs to the new release. For example, see https://review.opendev.org/709145.

RC2+

Further release candidates may be created on the stable branch as necessary in a similar manner to RC1.

Final Releases

A release candidate may be promoted to a final release if it has no critical bugs against it.

Tags should be created for each deliverable (`kayobe`, `kayobe-config`, `kayobe-config-dev`). Currently the same version is used for each.

The changes should be proposed to the releases repository. For example: https://review.opendev.org/701724.

Post-release activites

An email will be sent to the release-announce mailing list about the new release.

Continuing Development

Search for TODOs in the codebases describing tasks to be performed during the next release cycle.

Stable Releases

Stable branch releases should be made periodically for each supported stable branch, no less than once every 45 days.