TripleO Documentation

Release 0.0.1.dev1714

OpenStack Foundation

Aug 22, 2023

CONTENTS

TripleO is a project aimed at installing, upgrading and operating OpenStack clouds using OpenStacks own cloud facilities as the foundation - building on Nova, Ironic, Neutron and Heat to automate cloud management at datacenter scale

CHAPTER

ONE

CONTRIBUTOR GUIDE

1.1 TripleO Contributor Guide

1.1.1 Information for New Developers

The intention of this document is to give new developers some information regarding how to get started with TripleO as well as some best practices that the TripleO community has settled on.

In general TripleO is a very complex chunk of software. It uses numerous technologies to implement an OpenStack installer. The premise of TripleO was to use the OpenStack platform itself as the installer and API for user interfaces. As such the first step to installing TripleO is to create what is called an *undercloud*. We use almost similar architecture for both *undercloud* and *overcloud* that leverages same set of Heat templates found in *tripleo-heat-templates* repository, with a few minor differences. The *undercloud* services are deployed in containers and can be managed by the same tool chain used for *overcloud*.

Once the *undercloud* is deployed, we use a combination of Ansible playbooks and a set of Heat templates, to drive the deployment of an overcloud. Ironic is used to provision hardware and boot an operating system either on baremetal (for real deployments) or on VMs (for development). All services are deployed in containers on the overcloud like undercloud.

Repositories that are part of TripleO

- tripleo-common: This is intended to be for TripleO libraries of common code. Unfortunately it has become a bit overrun with unrelated bits. Work is ongoing to clean this up and split this into separate repositories.
- tripleo-ansible: Contains Ansible playbooks, roles, plugins, modules, filters for use with TripleO deployments.
- tripleo-heat-templates: This contains all the Heat templates necessary to deploy the overcloud (and hopefully soon the undercloud as well).
- python-tripleoclient: The CLI for deploying TripleO. This contains some logic but remember that we want to call Mistral actions from here where needed so that the logic can be shared with the UI.
- tripleo-docs: Where these docs are kept.
- tripleo-image-elements: Image elements (snippets of puppet that prepare specific parts of the image) for building the undercloud and overcloud disk images.
- tripleo-puppet-elements: Puppet elements used to configure and deploy the overcloud. These used during installation to set up the services.

- puppet-tripleo: Puppet is used to configure the services in TripleO. This repository contains various puppet modules for doing this.
- tripleo-quickstart: Quickstart is an Ansible driven deployment for TripleO used in CI. Most developers also use this to stand up instances for development as well.
- tripleo-quickstart-extras: Extended functionality for tripleo-quickstart allowing for end-to-end deployment and testing.
- tripleo-ui: The web based graphical user interface for deploying TripleO.
- kolla: We use the containers built by the Kolla project for services in TripleO. Any new containers or additions to existing containers should be submitted here.
- diskimage-builder: Disk image builder is used to build our base images for the TripleO deployment.

Definition of Done

This is basically a check list of things that you want to think about when implementing a new feature.

- Ensure that the continuous integration (CI) is in place and passing, adding coverage to tests if required. See http://specs.openstack.org/openstack/tripleo-specs/specs/policy/adding-ci-jobs.html for more information.
- Ensure there are unit tests where possible.
- Maintain backwards compatibility with our existing template interfaces from tripleo-heat-templates.
- New features should be reviewed by cores who have knowledge in that area of the codebase.
- One should consider logging and support implications. If you have new logs, would they be available via sosreport.
- Error messages are easy to understand and work their way back to the user (stack traces are not sufficient).
- Documentation should be updated if necessary. New features need a tripleo-docs patch.
- If any new dependencies are used for your feature, be sure they are properly packaged and available in RDO. You can ask on #rdo (on OFTC server) for help with this.

Using TripleO Standalone for Development

The Standalone container based deployment can be used for development purposes. This reuses the existing TripleO Heat Templates, allowing you to do the development using this framework instead of a complete overcloud. This is very useful if you are developing Heat templates or containerized services.

Please see Standalone Deployment Guide on how to set up a Standalone OpenStack node.

1.1.2 How to Contribute

TripleO source code is publicly available. You can contribute code to individual projects, documentation, report bugs and vulnerabilities, request features.

Contributing Code

As long as TripleO is a set of integrated OpenStack projects, all development is happening in OpenStack upstream.

Learn how to contribute into OpenStacks upstream.

See *TripleO Components* to find out how to contribute into individual projects.

Contacting the Core Team

Please refer to the TripleO Core Team contacts.

For upgrade specific contacts, refer to TripleO Upgrade Core contacts

For TripleO Ansible specific contacts, refer to TripleO Ansible Core contacts

For Shared TripleO CI role contacts, refer to TripleO Shared CI Core contacts

Contributing to this Documentation

TripleO User Documentation lives on git.opendev.org and is mirrored on GitHub under the OpenStack organization.

Learn how to contribute into TripleO Docs.

Reporting Bugs

OpenStack Upstream: If you find bugs or vulnerabilities which affect upstream projects, please follow OpenStacks process of filing bugs.

- Learn how to report bugs in OpenStack.
- If you want to file a bug against upstream project, you can find useful links in our list of *TripleO Components*.

TripleO If the bug impacts the TripleO project as a whole, you can file a bug in Launchpad:

1. Go to https://launchpad.net/tripleo
2. Fill in needed information (If you filed also upstream bug, please provide its URL in advanced fields)
3. Submit bug

Requesting Features

OpenStack Upstream: Since we are developing projects in OpenStack community, all the features are being requested upstream via Blueprints.

- Learn how to create Blueprints in OpenStack.
- If you want to file a bug against upstream project, you can find useful links in our list of *TripleO Components*.

1.1.3 Core maintainers

The intention of this document is to give developers some information regarding what is expected from core maintainers and hopefully provide some guidance to those aiming for this role.

Teams

The TripleO Core team is responsible for reviewing all changes proposed to repositories that are under the governance of TripleO.

The TripleO Upgrade core reviewers maintain the tripleo_upgrade project.

The TripleO Validation team maintains the Validation Framework in TripleO.

The TripleO CI team maintains the TripleO CI related projects (tripleo-ci, tripleo-quickstart, tripleo-quickstart-extras, etc).

We also have contributors with a specific area of expertise who have been granted core reviews on their area. Example: a Ceph integration expert would have core review on the Ceph related patches in TripleO.

Because Gerrit doesnt allow such granularity, we trust people to understand which patches they can use their core reviewer status or not. If one is granted core review access on an area, there is an expectation that itll only be used in this specific area. The grant is usually done for all the TripleO repositories but we expect SME cores to use +/- 2 for their area of expertise otherwise the regular +/- 1.

Note: Everyone is warmly encouraged to review incoming patches in TripleO, even if youre not (yet) a member of these teams. Participating in the review process will be a major task on the road to join the core maintainer teams.

Adding new members

Each team mentioned above should be aware of who is active in their respective project(s).

In order to add someone in one of these groups, it has to be discussed between other cores and the TripleO PTL.

It is a good practice to reach out to the nominee before proposing the candidate, to make sure about their willingness to accept this position and its responsibilities.

In real life, it usually happens by informal discussions, but the official proposals have to be sent with an email to the openstack-discuss mailing list. It is strongly recommended to have this initial informal agreement before going public, in case there are some disagreements which could cause unpleasant discussions which could harm the nominee.

This discussion can be initiated by any core, and only the existing cores votes will weight into whether or not the proposal is granted. Of course anyone is welcome to share their feedback and opinions.

Removing members

It is normal for developers to reduce their activity and work on something else. If they dont reach out by themselves, it is the responsibility of the teams to remove them from the core list and inform about the change on the mailing-list and privately when possible.

Also if someone doesnt respect the TripleO rules or doesnt use the core permission correctly, this person will be removed from the core list with a private notice at least.

Core membership expectations

Becoming a core member is a serious commitment and it is not granted easily. Here are a non-exhaustive list of things that are expected:

- The time invested on the project is consistent.
- (Nearly) Daily participation in core reviews.

Note: Core reviewers are expected to provide thoroughly reviews on the code, which doesnt only mean +1/-1, but also comments the code that confirm that the patch is ready (or not) to be merged into the repository. This capacity to provide these kind of reviews is strongly evaluated when recruiting new core reviewers. It is preferred to provide quality reviews over quantity. A negative review needs productive feedback and harmful comments wont help to build credibility within the team.

- Quality of technical contributions: bug reports, code, commit messages, specs, e-mails, etc.
- Awareness of discussions happening within the project (mailing-list, specs).
- Best effort participation on IRC #tripleo (when timezone permits), to provide support to our dear users and developers.
- Gain trust with other core members, engage collaboration and be nice with people. While mainly maintained by Red Hat, TripleO remains a friendly project where we hope people can have fun while maintaining a project which meets business needs for the OpenStack community.
- Understand the Expedited Approvals policy.

Final note

The goal of becoming core must not be intimidating. It should be reachable to anyone well involved in our project with has good intents and enough technical level. One should never hesitate to ask for help and mentorship when needed.

1.1.4 Squads

Work in TripleO is divided in Squads. For more information the project policy.

The list tends to be dynamic over the cycles, depending on which topics the team is working on. The list below is subject to change as squads change.

Squad	Description
CI	Group of people focusing on Continuous Integration tooling and system https://etherpad.openstack.org/p/tripleo-ci-squad-meeting
UI/CLI	Group of people focusing on TripleO UI and CLI https://etherpad.openstack.org/p/tripleo-ui-cli-squad-status
Upgrade	Group of people focusing on TripleO upgrades https://etherpad.openstack.org/p/tripleo-upgrade-squad-status
Validations	Group of people focusing on TripleO validations tooling https://etherpad.openstack.org/p/tripleo-validations-squad-status
Workflows	Group of people focusing on TripleO Workflows https://etherpad.openstack.org/p/tripleo-workflows-squad-status
Containers	Group of people focusing on TripleO deployed in containers https://etherpad.openstack.org/p/tripleo-containers-squad-status
Networking	Group of people focusing on networking bits in TripleO https://etherpad.openstack.org/p/tripleo-networking-squad-status
Integration	Group of people focusing on configuration management (eg: services) https://etherpad.openstack.org/p/tripleo-integration-squad-status
Edge	Group of people focusing on Edge/multi-site/multi-cloud https://etherpad.openstack.org/p/tripleo-edge-squad-status

Note: Note about CI: the squad is about working together on the tooling used by OpenStack Infra to test TripleO, though every squad has in charge of maintaining the good shape of their tests.

1.2 Developer Documentation

Documentation of developer-specific options in TripleO.

1.2.1 Composable services tutorial

This guide will be a walkthrough related to how to add new services to a TripleO deployment through additions to the tripleo-heat-templates and puppet-tripleo repositories, using part of the architecture defined in the composable services architecture.

Note: No puppet manifests may be defined in the THT repository, they should go to the puppet-tripleo repository instead.

Introduction

The initial scope of this tutorial is to create a brief walkthrough with some guidelines and naming conventions for future modules and features aligned with the composable services architecture. Regarding the example described in this tutorial, which leads to align an _existing_ non-composable service implementation with the composable roles approach, it is important to notice that a similar approach would be followed if a user needed to add an entirely new service to a tripleo deployment.

The puppet manifests used to configure services on overcloud nodes currently reside in the tripleo-heat-templates repository, in the folder puppet/manifests. In order to properly organize and structure the code, all manifests will be re-defined in the puppet-tripleo repository, and adapted to the composable services architecture.

The use case for this example uses NTP as a service installed by default among the OpenStack deployment. So the profile needs to be added to all the roles in roles_data.yaml.

Which means that NTP will be installed everywhere in the overcloud, so the tutorial will describe the process of refactoring the code from those files in order move it to the puppet-tripleo repository.

This tutorial is divided into several steps, according to different changes that need to be added to the structure of tripleo-heat-templates and puppet-tripleo.

Relevant repositories in this guide

- tripleo-heat-templates: All the tripleo-heat-templates (aka THT) logic.
- puppet-tripleo: TripleO puppet manifests used to deploy the overcloud services.
- tripleo-puppet-elements: References puppet modules used by TripleO to deploy the overcloud services. (Not used in this tutorial)

Gerrit patches used in this example

The gerrit patches used to describe this walkthrough are:

- https://review.opendev.org/#/c/310725/ (puppet-tripleo)
- https://review.opendev.org/#/c/310421/ (tripleo-heat-templates controller)
- https://review.opendev.org/#/c/330916/ (tripleo-heat-templates compute)
- https://review.opendev.org/#/c/330921/ (tripleo-heat-templates cephstorage)
- https://review.opendev.org/#/c/330923/ (tripleo-heat-templates objectstorage)

Change prerequisites

The controller services are defined and configured via Heat resource chains. In the proposed patch (https://review.opendev.org/#/c/259568) controller services will be wired to a new Heat feature that allows it to dynamically include a set of nested stacks representing individual services via a Heat resource chain. The current example will use this interface to decompose the controller role into isolated services.

Updating tripleo-heat-templates

This section will describe the changes needed for tripleo-heat-templates.

Folder structure convention for tripleo-heat-templates

Services should be defined in the services folder, depending on the service purpose.

```
puppet
  services          ---> To host all services.
    <service type>             ---> Folder to store a specific type services␣
↪(If time, will store time based services like: NTP, timezone, Chrony among␣
↪others).
      <service name>.yaml      ---> Heat template defining per-service␣
↪configuration.
      <service name>-base.yaml ---> Heat template defining common service␣
↪configuration.
```

Note: No puppet manifests may be defined in the THT repository, they should go to the puppet-tripleo repository instead.

Note: The use of a base heat template (<service>-base.yaml) is necessary in cases where a given service (e.g. heat) is comprised of a number of individual component services (e.g. heat-api, heat-engine) which need to share some of the base configuration (such as rabbit credentials). Using a base template in those cases means we dont need to duplicate that configuration. Refer to: https://review.opendev.org/#/c/313577/ for further details. Also, refer to *Duplicated parameters* for an use-case description.

Changes list

The list of changes in THT are:

- If there is any configuration of the given feature/service in any of the `tripleo-heat-templates/puppet/manifests/*.pp` files, then this will need to be removed and migrated to the puppet-tripleo repository.
- Create a service type specific folder in the root services folder (`deployment/time`).
- Create a heat template for the service inside the deployment/time folder (`deployment/time/ntp-baremetal-puppet.yaml`).

- Optionally, create a common heat template to reuse common configuration data, which is referenced from each per-service heat template.

Step 1 - Updating puppet references

Remove all puppet references for the composable service from the current manifests (*.pp). All the puppet logic will live in the puppet-tripleo repository based on a configuration step, so it is mandatory to remove all the puppet references from tripleo-heat-templates.

The updated .pp files for the NTP example were:

- `puppet/manifests/overcloud_cephstorage.pp`
- `puppet/manifests/overcloud_compute.pp`
- `puppet/manifests/overcloud_controller.pp`
- `puppet/manifests/overcloud_controller_pacemaker.pp`
- `puppet/manifests/overcloud_object.pp`
- `puppet/manifests/overcloud_volume.pp`

Step 2 - overcloud-resource-registry-puppet.j2.yaml resource registry changes

The resource `OS::TripleO::Services::Timesync` must be defined in the resource registry (`overcloud-resource-registry-puppet.j2.yaml`)

Create a new resource type alias which references the per-service heat template file, as described above.

By updating the resource registry we are forcing to use a nested template to configure our resources. In the example case the created resource (OS::TripleO::Services::Timesync), will point to the corresponding service yaml file (deployment/time/ntp-baremetal-puppet.yaml).

Step 3 - roles_data.yaml initial changes

The default roles are defined here. They are then iterated and the respective values of each section are rendered into the overcloud.j2.yaml.

Mandatory services should be added to the roles ServicesDefault value, which defines all the services enabled by default in the role(s).

From `roles_data.yaml` find:

```
- name: Controller
  CountDefault: 1
  ServicesDefault:
    - OS::TripleO::Services::CACerts
    - OS::TripleO::Services::CertmongerUser
    - OS::TripleO::Services::CephMds
    - OS::TripleO::Services::Keystone
    - OS::TripleO::Services::GlanceApi
    - OS::TripleO::Services::GlanceRegistry
```

(continues on next page)

(continued from previous page)

```
    ...
    - OS::TripleO::Services::Timesync              ---> New service deployed␣
↪in the controller overcloud
```

Update this section with your new service to be deployed to the controllers in the overcloud.

These values will be used by the controller roles ServiceChain resource as follows:

```
{% for role in roles %}
  # Resources generated for {{role.name}} Role
  {{role.name}}ServiceChain:
    type: OS::TripleO::Services
    properties:
      Services:
        get_param: {{role.name}}Services
      ServiceNetMap: {get_attr: [ServiceNetMap, service_net_map]}
      EndpointMap: {get_attr: [EndpointMap, endpoint_map]}


...
{% endfor %}
```

THT changes for all the different roles are covered in:

- https://review.opendev.org/#/c/310421/ (tripleo-heat-templates controller)
- https://review.opendev.org/#/c/330916/ (tripleo-heat-templates compute)
- https://review.opendev.org/#/c/330921/ (tripleo-heat-templates cephstorage)
- https://review.opendev.org/#/c/330923/ (tripleo-heat-templates objectstorage)

Note: In the case of the controller services, they are defined as part of the roles ServiceChain resource. If it is needed to add optional services, they need to be appended to the current services list defined by the default value of the roles ServicesDefault parameter.

Step 4 - Create the services yaml files

Create: `deployment/time/ntp-baremetal-puppet.yaml`

This file will have all the configuration details for the service to be configured.

```
heat_template_version: rocky
description: >
  NTP service deployment using puppet, this YAML file
  creates the interface between the HOT template
  and the puppet manifest that actually installs
  and configure NTP.
parameters:
  EndpointMap:
    default: {}
```

(continues on next page)

(continued from previous page)

```
    description: Mapping of service endpoint -> protocol. Typically set
                 via parameter_defaults in the resource registry.
    type: json
  NtpServers:
    default: ['0.pool.ntp.org', '1.pool.ntp.org']
    description: NTP servers
    type: comma_delimited_list
  NtpInterfaces:
    default: ['0.0.0.0']
    description: Listening interfaces
    type: comma_delimited_list
outputs:
  role_data:
    description: Role ntp using composable services.
    value:
      config_settings:
        ntp::ntpservers: {get_param: NtpServers}
        ntp::ntpinterfaces: {get_param: NtpInterfaces}
      step_config: |
        include ::tripleo::profile::base::time::ntp
```

Note: All role-specific parameters have to be tagged:

```
ExampleParameter:
  description: This is an example.
  type: json
  default: {}
  tags:
    - role_specific
```

Note: It is required for all service templates to accept the EndpointMap parameter, all other parameters are optional and may be defined per-service. Care should be taken to avoid naming collisions between service parameters, e.g via using the service name as a prefix, Ntp in this example.

Service templates should output a role_data value, which is a mapping containing config_settings which is a mapping of hiera key/value pairs required to configure the service, and step_config, which is a puppet manifest fragment that references the puppet-tripleo profile that configures the service.

If it is needed, the templates can be decomposed to remove duplicated parameters among different deployment environments (i.e. using pacemaker). To do this see section *Duplicated parameters*.

If your service has configuration that affects another service and should only be run on nodes (roles) that contain that service, you can use service_config_settings. You then have to specify the hieradata inside this section by using the name of the service as the key. So, if you want to output hieradata related to your service, on nodes that deploy keystone, you would do this:

```
role_data:
  ...
```

(continues on next page)

(continued from previous page)

```
step_config:
  ...
...
service_config_settings:
  keystone:
    # Here goes the hieradata
```

This is useful for things such as creating the keystone endpoints for your service, since one usually wants these commands to only be run on the keystone node.

Updating puppet-tripleo

The puppet manifests that currently define overcloud node configuration are moved from the tripleo-heat-templates to new puppet-tripleo class definitions as part of the composable services approach. In next iterations, all service configuration should be moved also to puppet-tripleo. This section considers the addition of the ntp definition to puppet-tripleo.

Folder structure convention

Services should be defined in the services folder, depending on the service purpose.

```
manifests
  profile/base      ---> To host all services not using pacemaker.
    time            ---> Specific folder for time services (NTP, timezone,␣
↪Chrony among others).
      ntp.pp        ---> Puppet manifest to configure the service.
```

Note: For further information related to the current folders manifests structure refer to the puppet-tripleo repository.

Adding the puppet manifest

This step will reference how the puppet logic should be organized in puppet-tripleo.

Inside the manifests folder, add the service manifest following the folder structure (`manifests/profile/base/time/ntp.pp`) as:

```
class tripleo::profile::base::time::ntp (
  #We get the configuration step in which we can choose which steps to execute
  $step          = hiera('step'),
) {
  #step assigned for core modules.
  #(Check for further references about the configuration steps)
  #https://opendev.org/openstack/tripleo-heat-templates/src/branch/master/
↪puppet/services/README.rst
```

(continues on next page)

(continued from previous page)

```
  if ($step >= 2){
    #We will call the NTP puppet class and assign our configuration values.
    #If needed additional Puppet packages can be added/installed by using the␣
↪repo tripleo-puppet-elements
    if count($ntpservers) > 0 {
      include ::ntp
    }
  }
}
```

If users have followed all the previous steps, they should be able to configure their services using the composable services within roles guidelines.

THT design patterns

Duplicated parameters

Problem: When defining multiple related services, it can be necessary to define the same parameters (such as rabbit or DB credentials) in multiple service templates. To avoid this, it is possible to define a base heat template that contains the common parameters and config_settings mapping for those services that require it.

This pattern will describe how to avoid duplicated parameters in the THT yaml files.

`mongodb-base.yaml`: This file should have all the common parameters between the different environments (With pacemaker and without pacemaker).

```
heat_template_version: rocky
description: >
  Configuration details for MongoDB service using composable roles
parameters:
  MongoDbNoJournal:
    default: false
    description: Should MongoDb journaling be disabled
    type: boolean
  MongoDbIPv6:
    default: false
    description: Enable IPv6 if MongoDB VIP is IPv6
    type: boolean
  MongoDbReplset:
    type: string
    default: "tripleo"
outputs:
  role_data:
    description: Role data for the MongoDB base service.
    value:
      config_settings:
        mongodb::server::nojournal: {get_param: MongoDbNoJournal}
        mongodb::server::ipv6: {get_param: MongoDbIPv6}
        mongodb::server::replset: {get_param: MongoDbReplset}
```

In this way we will be able to reuse the common parameter among all the template files requiring it.

Referencing the common parameter:

`mongodb.yaml`: Will have specific parameters to deploy mongodb without pacemaker.

```
heat_template_version: rocky
description: >
  MongoDb service deployment using puppet
parameters:
  #Parameters not used EndpointMap
  EndpointMap:
    default: {}
    description: Mapping of service endpoint -> protocol. Typically set
                 via parameter_defaults in the resource registry.
    type: json
resources:
  MongoDbBase:
    type: ./mongodb-base.yaml
outputs:
  role_data:
    description: Service mongodb using composable services.
    value:
      config_settings:
        map_merge:
          - get_attr: [MongoDbBase, role_data, config_settings]
          - mongodb::server::service_manage: True
      step_config: |
        include ::tripleo::profile::base::database::mongodb
```

In this case mongodb.yaml is using all the common parameter added in the MongoDbBase resource.

If using the parameter EndpointMap in the base template, you must the pass it from the service file, and even if it is not used in the service template, it must still be defined.

In the service file:

```
parameters:
  EndpointMap:
    default: {}
    description: Mapping of service endpoint -> protocol. Typically set
                 via parameter_defaults in the resource registry.
    type: json
resources:
  <ServiceName>ServiceBase:
    type: ./<ServiceName>-base.yaml
    properties:
      EndpointMap: {get_param: EndpointMap}
```

This will pass the endpoint information to the base config file.

Note: Even if the EndpointMap parameter is optional in the base template, for consistency is advised

always using it in all service templates.

TLS support for services

Public TLS

If youre adding a REST service to TripleO, chances are that youll need your service to be terminated by HAProxy. Unfortunately, adding your service to HAProxy needs extra changes to existing modules. Fortunately, its not that hard to do.

You can add your service to be terminated by HAproxy by modifying the manifests/haproxy.pp file.

First off, we need a flag to tell the HAProxy module to write the frontend for your service in the HAProxy configuration file if your service is deployed. For this, we will add a parameter for the manifest. If you have followed the walk-through, you may have noticed that the tripleo-heat-templates yaml template requires you to set a name for your service in the `role_data` output:

```
...
outputs:
  role_data:
    description: Description of your service
    value:
      service_name: my_service
...
```

The overcloud stack generated from the tripleo-heat-templates will use this name and automatically generate several hieradata entries that are quite useful. One of this entries is a global flag that can tell if your service is enabled at all or not. So well use this flag and fetch it from hiera to set the parameter we need in haproxy.pp:

```
...
$keystone_admin               = hiera('keystone_enabled', false),
$keystone_public              = hiera('keystone_enabled', false),
$neutron                      = hiera('neutron_api_enabled', false),
$cinder                       = hiera('cinder_api_enabled', false),
$glance_api                   = hiera('glance_api_enabled', false),
...
$my_service                   = hiera('my_service_enabled', false),
...
```

Note that the name of the hiera key matches the following format <service name>_enabled and defaults to `false`.

Next, you need to add a parameter that tells HAProxy which network your service is listening on:

```
...
$barbican_network             = hiera('barbican_api_network', false),
$ceilometer_network           = hiera('ceilometer_api_network', undef),
$cinder_network               = hiera('cinder_api_network', undef),
$glance_api_network           = hiera('glance_api_network', undef),
$heat_api_network             = hiera('heat_api_network', undef),
```

(continues on next page)

(continued from previous page)

```
...
$my_service_network          = hiera('my_service_network', undef),
...
```

Tripleo-heat-templates will also autogenerate this key for you. However for it to do this, you need to specify the network for your service in the templates. The file where this needs to be set is network/service_net_map.j2.yaml, and youll be looking for a parameter called `ServiceNetMapDefaults`. It will look like this:

```
# Note that the key in this map must match the service_name
# see the description above about conversion from CamelCase to
# snake_case - the names must still match when converted
ServiceNetMapDefaults:
  default:
    # Note the values in this map are replaced by *NetName
    # to allow for sane defaults when the network names are
    # overridden.
    ...
    NeutronTenantNetwork: tenant
    CeilometerApiNetwork: internal_api
    BarbicanApiNetwork: internal_api
    CinderApiNetwork: internal_api
    GlanceApiNetwork: storage
    ...
    MyServiceNetwork: <some network>
```

Now, having added this, youll have access to the aforementioned hiera key and several others.

Note that the network is used by HAProxy to terminate TLS for your service. This is used when Internal TLS is enabled and youll learn more about it in the *Internal TLS* section.

Then, you need to add the ports that HAProxy will listen on. There is a list with the defaults which is called `default_service_ports`, and you need to add your service here:

```
$default_service_ports = {
  ...
  neutron_api_port => 9696,
  neutron_api_ssl_port => 13696,
  nova_api_port => 8774,
  nova_api_ssl_port => 13774,
  nova_placement_port => 8778,
  nova_placement_ssl_port => 13778,
  nova_metadata_port => 8775,
  nova_novnc_port => 6080,
  nova_novnc_ssl_port => 13080,
  ...
  my_service_port => 5123,
  my_service_ssl_port => 13123,
  ...
}
```

You are specifying two ports here, one that is the standard port, and another one that is used for SSL in

the public VIP/host. This was done initially to address deployments without network isolation. In these cases, deploying TLS would effectively take over the other interfaces, so HAProxy would be listening with TLS everywhere accidentally if only using one port, and further configuration for the services would need to happen to address this. However, this is not really an issue in network isolated deployments, since they would be using different IP addresses. So this extra port might not be needed in the future if network isolation becomes the standard mode of deploying.

Note: The SSL port is not needed if your service is only internal and doesnt listen on the public VIP.

Note: These ports can be overwritten by using the `$service_ports` parameter from this manifest. Once could pass it via hieradata through the `ExtraConfig` tripleo-heat-templates parameter, and setting something like this as the value:

```
tripleo::haproxy::service_ports:
  my_service_ssl_port: 5123
  my_service_2_ssl_port: 5124
```

Please consider that this will overwrite any entry from the list of defaults, so you have to be careful to update all the relevant entries in tripleo-heat-templates if you want to change port (be it SSL port or non-SSL port).

Finally, you need to add the actual endpoint to HAproxy which will configure the listen directive (or frontend and backend) in the haproxy configuration. For this, we have a helper class called `::tripleo::haproxy::endpoint` that sets the relevant bits for you. All we need to do is pass in all the information that class needs. And we need to make sure that this only happens if the service is enabled, so well enclose it with the flag we mentioned above. So heres a code snippet that demonstrates what you need to add:

```
if $my_service {
  ::tripleo::haproxy::endpoint { 'my_service':
    public_virtual_ip => $public_virtual_ip,
    internal_ip       => hiera('my_service_vip', $controller_virtual_ip),
    service_port      => $ports[my_service_port],
    ip_addresses      => hiera('my_service_node_ips', $controller_hosts_real),
    server_names      => hiera('my_service_node_names', $controller_hosts_
↪names_real),
    mode              => 'http',
    listen_options    => {
        'http-request' => [
          'set-header X-Forwarded-Proto https if { ssl_fc }',
          'set-header X-Forwarded-Proto http if !{ ssl_fc }'],
    },
    public_ssl_port   => $ports[my_service_ssl_port],
    service_network   => $my_service_network,
  }
}
```

- The `public_virtual_ip` variable contains the public IP address thats used for your cloud, and its the one that people will usually have access to externally.

- The hiera keys `my_service_node_ips`, `my_service_vip`, `my_service_node_names` are automatically generated by tripleo-heat-templates. These are other keys that youll get access to once you add the network for your service in `ServiceNetMapDefaults`.
- `my_service_vip` is, as mentioned, automatically generated, and will point HAProxy to the non-public VIP where other services will be able to access your service. This will usually be the Internal API network, but it depends on your use-case.
- `my_service_node_ips` is, as mentioned, automatically generated, and will tell HAProxy which nodes are hosting your service, so it will point to them. The address depends on the network your service is listening on.
- `my_service_node_names` is, as mentioned, automatically generated, and will be the names that HAProxy will use for the nodes. These are the FQDNs of the nodes that are hosting your service.
- This example is an HTTP service, so note that we set the mode to `http`, and that we set the option for HAProxy to detect if TLS was used for the request, and set an appropriate value for the `X-Forwarded-Proto` HTTP header if thats the case. Not all services can read this HTTP header, so this depends on your service. For more information on the available options and the mode, consult the haproxy documentation.

Note: If your service is only internal and doesnt listen on the public VIP, you dont need all of the parameters listed above, and you would instead do something like this:

```
if $my_service {
  ::tripleo::haproxy::endpoint { 'my_service':
    internal_ip     => hiera('my_service_vip', $controller_virtual_ip),
    service_port    => $ports[my_service_port],
    ip_addresses    => hiera('my_service_node_ips', $controller_hosts_real),
    server_names    => hiera('my_service_node_names', $controller_hosts_names_
↪real),
    service_network => $my_service_network,
  }
}
```

The most relevant bits are that we omitted the SSL port and the `public_virtual_ip`, since these wont be used.

Having added this to the manifest, you should be covered for both getting your service to be proxied by HAProxy, and letting it to TLS in the public interface for you.

Internal TLS

How it works

If you havent read the section TLS Everywhere it is highly recommended you read that first before continuing.

As mentioned, the default CA is FreeIPA, which issues the certificates that the nodes request, and they do the requests via certmonger.

FreeIPA needs to have the nodes registered in its database and those nodes need to be enrolled in order to authenticate to the CA. This is already being handled for us, so theres nothing you need to do for your service on this side.

In order to issue certificates, FreeIPA also needs to have registered a Kerberos principal for the service (or service principal). This way it knows what service is using what certificate. The service principal will look something like this:

```
<service name>/<host>.<domain>
```

We assume that the domain matches the kerberos realm, so specifying it is redundant.

Fortunately, one doesnt need to do much but fill in some boilerplate code in tripleo-heat-templates to get this service principal. And this will be covered in subsequent sections.

So, with this one can finally request certificates for the service and use them.

Enabling internal TLS for your service

Aside from the actual certificate request, if your service is a RESTful service, getting TLS to work with the current solution requires usually two fronts:

- To get your service to actually serve with TLS.
- To tell HAProxy to try to access your service using TLS.

This can be different for other types of services. For instance, at the time of writing this, RabbitMQ isnt proxied by HAProxy, so there wasnt a need to configure anything in HAProxy. Another example is MariaDB: Even though it is proxied by HAProxy, TLS is handled on the MariaDB side and HAProxy doesnt do TLS termination, so there was no need to configure HAProxy.

Also, for services in general, there are two options for the Subject Alternative Name (SAN) for the certificate:

1) It should be a hostname that points to a specific interface in the node.

2) It should be a hostname that points to a VIP (or a Virtual IP Address).

The usual case for a RESTful service will be the first option. HAProxy will do TLS termination, listening on the clouds VIPs, and will then forward the request to your service trying to access it via the nodes internal network interface (not the VIP). So for this case (#1), your service should be serving a TLS certificate with the nodes interface as the SAN. RabbitMQ has a similar situation even if its not proxied by HAProxy. Services try to access the RabbitMQ cluster through the individual nodes, so each broker server has a certificate with the nodes hostname for a specific network interface as the SAN. On the other hand, MariaDB follows the SAN pattern #2. Its terminated by HAProxy, so the services access it through a VIP. However, MariaDB handles TLS by itself, so it ultimately serves certificates with the hostname pointing to a VIP interface as the SAN. This way, the hostname validation works as expected.

If youre not sure how to go forward with your service, consult the TripleO team.

Services that run over httpd

Good news! Certificates are already requested for you and there is a hash where you can fetch the path to the certificates and use them for your service.

In puppet-tripleo you need to go to the manifest that deploys the API for your service. Here, you will add the following parameters to the class:

```
class tripleo::profile::base::my_service::api (
  ...
  $my_service_network  = hiera('my_service_network', undef),
  $certificates_specs  = hiera('apache_certificates_specs', {}),
  $enable_internal_tls = hiera('enable_internal_tls', false),
  ...
) {
```

- `my_service_network` is a hiera key thats already generated by tripleo-heat-templates and it references the name of the network your service is listening on. This was referenced in the *Public TLS* section. Where it mentioned the addition of your services network to the `ServiceNetMapDefaults` parameter. So, if this was done, youll get this key autogenerated.
- `apache_certificates_specs` is a hash containing the specifications for all the certificates requested for services running over httpd. These are network-dependant, which is why we needed the network name. Note that this also contains the paths where the keys are located in the filesystem.
- `enable_internal_tls` is a flag that tells TripleO if TLS for the internal network is enabled. We should base the usage of the certificates for your service on this.

In order to get the certificate and key for your application you can use the following boilerplate code:

```
if $enable_internal_tls {
  if !$my_service_network {
    fail('my_service_network is not set in the hieradata.')
  }
  $tls_certfile = $certificates_specs["httpd-${my_service_network}"]['service_
↪certificate']
  $tls_keyfile = $certificates_specs["httpd-${my_service_network}"]['service_
↪key']
} else {
  $tls_certfile = undef
  $tls_keyfile = undef
}
```

If internal TLS is not enabled, we set the variables for the certificate and key to `undef`, this way TLS wont be enabled. If its enabled, we get the certificate and key from the hash.

Now, having done this, we can pass in the variables to the class that deploys your service over httpd:

```
class { '::my_service::wsgi::apache':
  ssl_cert => $tls_certfile,
  ssl_key  => $tls_keyfile,
}
```

Now, in tripleo-heat-templates, hopefully the template for your services API already uses the base profile

for apache services. To verify this, you need to look in the `resources` section of your template for something like this:

```
ApacheServiceBase:
  type: ./apache.yaml
  properties:
    ServiceNetMap: {get_param: ServiceNetMap}
    EndpointMap: {get_param: EndpointMap}
```

Note that this is of type ./apache.yaml which is the template that contains the common configurations for httpd based services.

You will also need to make sure that the ssl hieradata is set correctly. You will find it usually like this:

```
my_service::wsgi::apache::ssl: {get_param: EnableInternalTLS}
```

Where, EnableInternalTLS should be defined in the `parameters` section of the template.

Finally, you also need to add the `metadata_settings` to the output of the template. This section will be in the same level as `config_settings` and `step_config`, and will contain the following:

```
metadata_settings:
  get_attr: [ApacheServiceBase, role_data, metadata_settings]
```

Note that it merely outputs the metadata_settings section that the apache base stack already outputs. This will give the appropriate parameters to a hook that sets the nova metadata, which in turn will be taken by the *novajoin* service generate the service principals for httpd for the host.

See the TLS Everywhere Deploy Guide

Configuring HAProxy to use TLS for your service

Now that your service will be serving with TLS enabled, we go back to the manifests/haproxy.pp file. You already have added the HAProxy endpoint resource for your service, so for this, you need to add now the option to tell it to use TLS to communicate with the server backend nodes. This is done by adding this:

```
if $my_service {
  ::tripleo::haproxy::endpoint { 'my_service':
    ...
    member_options    => union($haproxy_member_options, $internal_tls_member_
↪options),
  }
}
```

This adds the TLS options to the default member options we use in TripleO for HAProxy. It will tell HAProxy to require TLS for your service if internal TLS is enabled; if its not enabled, then it wont use TLS.

This was all the extra configuration you needed to do for HAProxy.

Internal TLS for services that dont run over httpd

If your service supports being run with TLS enabled, and its not python/eventlet-based (see *Internal TLS via a TLS-proxy*). This section is for you.

In tripleo-heat-templates well need to specify the specs for doing the certificate request, and well need to get the appropriate information to generate a service principal. To make this optional, you should add the following to your services base template:

```
parameters:
  ...
  EnableInternalTLS:
    type: boolean
    default: false

conditions:

  internal_tls_enabled: {equals: [{get_param: EnableInternalTLS}, true]}
  ...
...
```

- `EnableInternalTLS` is a parameter thats passed via `parameter_defaults` which tells the templates that we want to use TLS in the internal network.
- `internal_tls_enabled` is a condition that well furtherly use to add the relevant bits to the output.

The next thing to do is to add the certificate specs, the relevant hieradata and the required metadata to the output. In the `roles_data` output, lets modify the `config_settings` to add what we need:

```
config_settings:
  map_merge:
    -
      # The regular hieradata for your service goes here.
      ...
    -
      if:
      - internal_tls_enabled
      - generate_service_certificates: true
        my_service_certificate_specs:
          service_certificate: '/etc/pki/tls/certs/my_service.crt'
          service_key: '/etc/pki/tls/private/my_service.key'
          hostname:
            str_replace:
              template: "%{hiera('fqdn_NETWORK')}"
              params:
                NETWORK: {get_param: [ServiceNetMap, MyServiceNetwork]}
          principal:
            str_replace:
              template: "my_service/%{hiera('fqdn_NETWORK')}"
              params:
                NETWORK: {get_param: [ServiceNetMap, MyServiceNetwork]}
      - {}
```

(continues on next page)

(continued from previous page)

```
...
metadata_settings:
  if:
    - internal_tls_enabled
    -
      - service: my_service
        network: {get_param: [ServiceNetMap, MyServiceNetwork]}
        type: node
    - null
```

- The conditional mentioned above is used in the `config_settings`. So, if `internal_tls_enabled` evaluates to `true`, the hieradata necessary to enable TLS in the internal network for your service will be added. Else, we output `{}`, which wont affect the `map_merge` and wont add anything to the regular hieradata for your service.
- For this case, we are only requesting one certificate for the service.
- The service will be terminated by HAProxy in a conventional way, which means that the SAN will be case #1 as described in *Enabling internal TLS for your service*. So the SAN will point to the specific nodes network interface, and not the VIP.
- The `ServiceNetMap` contains the references to the networks every service is listening on, and the key to get the network is the name of your service but using camelCase instead of underscores. This value is the name of the network and if used under the `config_settings` section, it will be replaced by the actual IP. Else, it will just be the network name.
- tripleo-heat-templates automatically generates hieradata that contains the different network-dependant hostnames. They keys are in the following format:

  ```
  fqdn_<network name>
  ```

- The `my_service_certificate_specs` key will contain the specifications for the certificate well request. They need to follow some conventions:
  - `service_certificate` will specify the path to the certificate file. It should be an absolute path.
  - `service_key` will specify the path to the private key file that will be used for the certificate. It should be an absolute path.
  - `hostname` is the name that will be used both in the Common Name (CN) and the Subject Alternative Name (SAN) of the certificate. We can get this value by using the hiera key described above. So we first get the name of the network the service is listening on from the `ServiceNetMap` and we then use `str_replace` to place that in a hiera call in the appropriate format.
  - `principal` is the service principal that will be the one used for the certificate request. We can get this in a similar manner as we got the hostname, and prepending an identifying name for your service. The format will be as follows:

    ```
    < service identifier >/< network-based hostname >
    ```

  - These are the names used by convention, and will eventually be passed to the `certmonger_certificate` resource from puppet-certmonger.

- The `metadata_settings` section will pass some information to a metadata hook that will create the service principal before the certificate request is done. The format as follows:
  - `service`: This contains the service identifier to be used in the kerberos service principal. It should match the identifier you put in the `principal` section of the certificate specs.
  - `network`: Tells the hook what network to use for the service. This will be used for the hook and novajoin to use an appropriate hostname for the kerberos principal.
  - `type`: Will tell the hook what type of case is this service. The available options are `node` and `vip`. These are the cases mentioned in the *Enabling internal TLS for your service* for the SANs.

  Note that this is a list, which can be useful if well be creating several service principals (which is not the case for our example). Also, if `internal_tls_enabled` evaluates to `false`, we then output `null`.
- Remember to set any relevant flags or parameters that your service might need as hieradata in `config_settings`. These might be things that explicitly enable TLS such as flags or paths. But these details depend on the puppet module that deploys your service.

Note: VIP-based hostname case

If your service requires the certificate to contain a VIP-based hostname, as is the case for MariaDB. It would instead look like the following:

```
metadata_settings:
  if:
    - internal_tls_enabled
    -
      - service: my_service
        network: {get_param: [ServiceNetMap, MyServiceNetwork]}
        type: vip
    - null
```

- One can get the hostname for the VIP in a similar fashion as we got the hostname for the node. The VIP hostnames are also network based, and one can get them from a hiera key as well. It has the following format:

  ```
  cloud_name_< network name >
  ```

- The `type` in the `metadata_settings` entry is `vip`.

In puppet-tripleo Well create a class that does the actual certificate request and add it to the resource that gets the certificates for all the services.

Lets create a class to do the request:

```
class tripleo::certmonger::my_service (
  $hostname,
  $service_certificate,
  $service_key,
  $certmonger_ca = hiera('certmonger_ca', 'local'),
  $principal     = undef,
```

(continues on next page)

(continued from previous page)

```
) {
  include ::my_service::params

  $postsave_cmd  = "systemctl restart ${::my_service::params::service_name}"
  certmonger_certificate { 'my_service' :
    ensure       => 'present',
    certfile     => $service_certificate,
    keyfile      => $service_key,
    hostname     => $hostname,
    dnsname      => $hostname,
    principal    => $principal,
    postsave_cmd => $postsave_cmd,
    ca           => $certmonger_ca,
    wait         => true,
    require      => Class['::certmonger'],
  }

  file { $service_certificate :
    owner   => $::my_service::params::user,
    group   => $::my_service::params::group,
    require => Certmonger_certificate['my_service'],
  }
  file { $service_key :
    owner   => $::my_service::params::user,
    group   => $::my_service::params::group,
    require => Certmonger_certificate['my_service'],
  }

  File[$service_certificate] ~> Service<| title == $::my_
↪service::params::service_name |>
  File[$service_key] ~> Service<| title == $::my_service::params::service_
↪name |>
}
```

- Youll note that the parameters mostly match the certificate specs that we created before in tripleo-heat-templates.
- By convention, well add this class in the **manifests/certmonger** folder.
- `certmonger_ca` is a value that comes from tripleo-heat-templates and tells certmonger which CA to use.
- If its available, by convention, many puppet modules contain a manifest called *params*. This usually contains the name and group that the service runs with, as well as the name of the service in a specific distribution. So we include this.
- We do then the actual certificate request by using the `certmonger_certificate` provider and passing all the relevant data for the request.
  - The post-save command which is specified via the `postsave_cmd` is a command that will be ran after the certificate is saved. This is useful for when certmonger has to resubmit the request to get an updated certificate, since this way we can reload or restart the service so it

can serve the new certificate.

- Using the `file` resource from puppet, we set the appropriate user and group for the certificate and keys. Fortunately, certmonger has sane defaults for the file modes, so we didnt set those here.

Having this class, we now need to add to the certmonger_user resource. This resource is in charge of making all the certificate requests and should be available on all roles (or at least it should be added). You would add the certificate specs as a parameter to this class:

```
class tripleo::profile::base::certmonger_user (
  ...
  $my_service_certificate_specs = hiera('my_service_certificate_specs', {}),
  ...
) {
```

And finally, we call the class that does the request:

```
...
unless empty($my_service_certificate_specs) {
  ensure_resource('class', 'tripleo::certmonger::my_service', $my_service_
↪certificate_specs)
}
...
```

Note: It is also possible to do several requests for your service. See the certmonger_user source code for examples.

Finally, you can do the same steps described in *configuring-haproxy-internal-tls* to make HAProxy connect to your service using TLS.

Internal TLS via a TLS-proxy

If you have a RESTful service that runs over python (most likely using eventlet) or if your service requires a TLS proxy in order to have TLS in the internal network, there are extra steps to be done.

For python-based services, due to performance issues with eventlet, the best thing you can do is try to move your service to run over httpd, and let it handle crypto instead. Then youll be able to follow the instructions from the *Services that run over httpd* section above. If for any reason this cant be done at the moment, we could still use httpd to service as a TLS proxy in the node. It would then listen on the services port and forward all the requests to the service, which would then be listening on localhost.

In puppet-tripleo you need to go to the manifest that deploys the API for your service, and add the following parameters:

```
class tripleo::profile::base::my_service::api (
  ...
  $certificates_specs  = hiera('apache_certificates_specs', {}),
  $enable_internal_tls = hiera('enable_internal_tls', false),
  $my_service_network  = hiera('my_service_api_network', undef),
  $tls_proxy_bind_ip   = undef,
  $tls_proxy_fqdn      = undef,
```

(continues on next page)

(continued from previous page)

```
  $tls_proxy_port      = 5123,
  ...
) {
...
```

- `certificates_specs`, `enable_internal_tls` and `my_service_network` have already been mentioned in the *Services that run over httpd* section.
- `tls_proxy_bind_ip`, `tls_proxy_fqdn` and `tls_proxy_port` are parameters that will be used by the httpd-based TLS proxy. They will tell it where what IP to listen on, the FQDN (which will be used as the servername) and the port it will use. Usually the port will match your services port. These values are expected to be set from tripleo-heat-templates.

Next comes the code for the actual proxy:

```
...
if $enable_internal_tls {
  if !$my_service_network {
    fail('my_service_network is not set in the hieradata.')
  }
  $tls_certfile = $certificates_specs["httpd-${my_service_network}"]['service_
↪certificate']
  $tls_keyfile = $certificates_specs["httpd-${my_service_network}"]['service_
↪key']

  ::tripleo::tls_proxy { 'my_service_proxy':
    servername => $tls_proxy_fqdn,
    ip         => $tls_proxy_bind_ip,
    port       => $tls_proxy_port,
    tls_cert   => $tls_certfile,
    tls_key    => $tls_keyfile,
    notify     => Class['::my_service::api'],
  }
}
...
```

- The `::tripleo::tls_proxy` is the resource that will configure the TLS proxy for your service. As you can see, it receives the certificates that come from the `certificates_specs` which contain the specification for the certificates, including the paths for the keys.
- The notify is added here since we want the proxy to be set before the service.

In tripleo-heat-templates, you should modify your services template and add the following:

```
parameters:
...
  EnableInternalTLS:
    type: boolean
    default: false
...
conditions:
  ...
```

(continues on next page)

(continued from previous page)

```
  use_tls_proxy: {equals : [{get_param: EnableInternalTLS}, true]}
...
resources:
...
  TLSProxyBase:
    type: OS::TripleO::Services::TLSProxyBase
    properties:
      ServiceNetMap: {get_param: ServiceNetMap}
      EndpointMap: {get_param: EndpointMap}
      EnableInternalTLS: {get_param: EnableInternalTLS}
```

- `EnableInternalTLS` is a parameter thats passed via `parameter_defaults` which tells the templates that we want to use TLS in the internal network.
- `use_tls_proxy` is a condition that well use to modify the behaviour of the template depending on whether TLS in the internal network is enabled or not.
- `TLSProxyBase` will make the default values from the proxys template available to where our service is deployed. We should make sure that we combine our services hieradata with the hieradata coming from that resource by doing a `map_merge` with the `config_settings`:

```
...
config_settings:
  map_merge:
    - get_attr: [TLSProxyBase, role_data, config_settings]
    - # Here goes our service's metadata
      ...
```

So, with this, we can tell the service to bind on localhost instead of the default interface depending if TLS in the internal network is enabled or not. Lets now set the hieradata that the puppet module needs in our services hieradata, which is in the `config_settings` section:

```
tripleo::profile::base::my_service::api::tls_proxy_bind_ip:
  get_param: [ServiceNetMap, MyServiceNetwork]
tripleo::profile::base::my_service::api::tls_proxy_fqdn:
  str_replace:
    template:
      "%{hiera('fqdn_$NETWORK')}"
    params:
      $NETWORK: {get_param: [ServiceNetMap, MyServiceNetwork]}
tripleo::profile::base::my_service::api::tls_proxy_port:
  get_param: [EndpointMap, NeutronInternal, port]
my_service::bind_host:
  if:
  - use_tls_proxy
  - 'localhost'
  - {get_param: [ServiceNetMap, MyServiceNetwork]}
```

- The `ServiceNetMap` contains the references to the networks every service is listening on, and the key to get the network is the name of your service but using camelCase instead of underscores. This value will be automatically replaced by the actual IP.

- tripleo-heat-templates generates automatically hieradata that contains the different network-dependant hostnames. They keys are in the following format:

  ```
  fqdn_<network name>
  ```

  So, to get it, we get the network name from the `ServiceNetMap`, and do a `str_replace` in heat that will use that network name and add it to a hiera call that will then gets us the FQDN we need.

- The port we can easily get from the `EndpointMap`.
- The conditional uses the actual IP if theres no TLS in the internal network enabled and localhost if it is.

Finally, we add the `metadata_settings` section to make sure we get a kerberos service principal:

```
metadata_settings:
  get_attr: [TLSProxyBase, role_data, metadata_settings]
```

Summary

References:

1. https://etherpad.openstack.org/p/tripleo-composable-roles-work
2. https://review.opendev.org/#/c/245804/2/specs/mitaka/composable-services-within-roles.rst
3. https://review.opendev.org/#/q/topic:composable_service
4. https://docs.openstack.org/tripleo-docs/latest/install/advanced_deployment/template_deploy.html
5. http://hardysteven.blogspot.com.es/2015/05/tripleo-heat-templates-part-1-roles-and.html
6. http://hardysteven.blogspot.com.es/2015/05/tripleo-heat-templates-part-2-node.html
7. http://hardysteven.blogspot.com.es/2015/05/tripleo-heat-templates-part-3-cluster.html

Service template sections description

As mentioned in the previous sections of the developer guide, there are several sections of the templates output that need to be filled out for creating a service in TripleO.

In this document we will attempt to enumerate all of them and explain the reasoning behind them.

Note that you can also find useful information in the tht deployment readme.

Whats the bare-minimum?

Before, digging into details, its always good to know what the bare-minimum is. So lets look at a very minimal service template:

```
heat_template_version: rocky

description: Configure Red Hat Subscription Management.
```

(continues on next page)

(continued from previous page)

```
parameters:
  RoleNetIpMap:
    default: {}
    type: json
  ServiceData:
    default: {}
    description: Dictionary packing service data
    type: json
  ServiceNetMap:
    default: {}
    description: Mapping of service_name -> network name. Typically set
                 via parameter_defaults in the resource registry.  This
                 mapping overrides those in ServiceNetMapDefaults.
    type: json
  RoleName:
    default: ''
    description: Role name on which the service is applied
    type: string
  RoleParameters:
    default: {}
    description: Parameters specific to the role
    type: json
  EndpointMap:
    default: {}
    description: Mapping of service endpoint -> protocol. Typically set
                 via parameter_defaults in the resource registry.
    type: json
  RhsmVars:
    default: {}
    description: Hash of ansible-role-redhat-subscription variables
                 used to configure RHSM.
    # The parameters contains sensible data like activation key or password.
    hidden: true
    tags:
      - role_specific
    type: json

resources:
  # Merging role-specific parameters (RoleParameters) with the default␣
↪parameters.
  # RoleParameters will have the precedence over the default parameters.
  RoleParametersValue:
    type: OS::Heat::Value
    properties:
      type: json
      value:
        map_replace:
          - map_replace:
```

(continues on next page)

(continued from previous page)

```
            - vars: RhsmVars
            - values: {get_param: [RoleParameters]}
          - values:
              RhsmVars: {get_param: RhsmVars}

outputs:
  role_data:
    description: Role data for the RHSM service.
    value:
      service_name: rhsm
      config_settings:
        tripleo::rhsm::firewall_rules: {}
      upgrade_tasks: []
      step_config: ''
      host_prep_tasks:
        - name: Red Hat Subscription Management configuration
          vars: {get_attr: [RoleParametersValue, value, vars]}
          block:
          - include_role:
              name: redhat-subscription
```

Lets go piece by piece and explain whats going on.

Version and description

As with any other heat template, you do need to specify the `heat_template_version`, and preferably give a description of what the stack/template does.

Parameters

Youll notice that there are a bunch of heat parameters defined in this template that are not necessarily used. This is because service templates are created in the form of a heat resource chain object. This type of objects can create a chain or a set of objects with the same parameters, and gather the outputs of them. So, eventually we pass the same mandatory parameters to the chain. This happens in the common/services.yaml file. Lets take a look and see how this is called:

```
ServiceChain:
  type: OS::Heat::ResourceChain
  properties:
    resources: {get_param: Services}
    concurrent: true
    resource_properties:
      ServiceData: {get_param: ServiceData}
      ServiceNetMap: {get_param: ServiceNetMap}
      EndpointMap: {get_param: EndpointMap}
      RoleName: {get_param: RoleName}
      RoleParameters: {get_param: RoleParameters}
```

Here we can see that the mandatory parameters for the services are the following:

- **ServiceData**: Contains an entry called `net_cidr_map`, which is a map that has the CIDRs for each network in your deployment.
- **ServiceNetMap**: Contains a mapping that tells you what network is each service configured at. Typical entries will look like: `BarbicanApiNetwork: internal_api`.
- **EndpointMap**: Contains the keystone endpoints for each service. With this youll be able to get what port, what protocol, and even different entries for the public, internal and admin endpoints.
- **RoleName**: This is the name of the role on which the service is applied. It could be one of the default roles (e.g. Controller or Compute), or a custom role, depending on how youre deploying.
- **RoleParameters**: A Map containing parameters to be applied to the specific role.

So, if youre writing a service template yourself, these are the parameters you have to copy into your template.

Aside from these parameters, you can define any other parameter yourself for the service, and in order for your service to consume the parameter, you need to pass them via `parameter_defaults`.

The `role_data` output

This is the sole output that will be read and parsed in order to get the relevant information needed from your service. Its value must be a map, and from the aforementioned example, it minimally contains the following:

- `service_name`: This is the name of the service youre configuring. The format is lower case letters and underscores. Setting this is quite important, since this is how TripleO reports what services are enabled, and generates appropriate hieradata, such as a list of all services enabled, and flags that say that your service is enabled on a certain node.
- `config_settings`: This will contain a map of key value pairs; the map will be written to the hosts in the form of hieradata, which puppet can then run and use to configure your service. Note that the hieradata will only be written on hosts that are tagged with a role that enables your service.
- `upgrade_tasks`: These are ansible tasks that run when TripleO is running an upgrade with your service enabled. If you dont have any upgrade tasks to do, you still have to specify this output, but its enough to set it as an empty list.
- `step_config`: This defines what puppet manifest should be run to configure your service. It typically is a string with the specific `include` statement that puppet will run. If youre not configuring your service with puppet, then you need to set this value as an empty string. There is an exception, however: When youre configuring a containerized service. Well dig into that later.

These are the bare-minimum sections of `role_data` you need to set up. However, you might have noticed that the example we linked above has another section called `host_prep_data`. This section is not mandatory, but it is one of the several ways you can execute Ansible tasks on the host in order to configure your service.

Ansible-related parameters

The following are sections of the service template that allow you to use Ansible to execute actions or configure your service.

Host prep deployment (or `host_prep_tasks`)

This is seen as `host_prep_tasks` in the deployment service templates. These are Ansible tasks that run before the configuration steps start, and before any major services are configured (such as pacemaker). Here you would put actions such as wiping out your disk, or migrating log files.

Lets look at the output section of the example from the previous blog post:

```
outputs:
  role_data:
    description: Role data for the RHSM service.
    value:
      service_name: rhsm
      config_settings:
        tripleo::rhsm::firewall_rules: {}
      upgrade_tasks: []
      step_config: ''
      host_prep_tasks:
        - name: Red Hat Subscription Management configuration
          vars: {get_attr: [RoleParametersValue, value, vars]}
          block:
          - include_role:
              name: redhat-subscription
```

Here we see that an Ansible role is called directly from the `host_prep_tasks` section. In this case, were setting up the Red Hat subscription for the node where this is running. We would definitely want this to happen in the very beginning of the deployment, so `host_prep_tasks` is an appropriate place to put it.

Pre Deploy Step tasks (or `pre_deploy_step_tasks`)

These are Ansible tasks that take place in the overcloud nodes. They are run after the network is completely setup, after the bits to prepare for containers running are completed (TCIB/Kolla files, container engine installation and configuration). They are also run before any External deploy tasks.

External deploy tasks

These are Ansible tasks that take place in the node where you executed the overcloud deploy. Youll find these in the service templates in the `external_deploy_tasks` section. These actions are also ran as part of the deployment steps, so youll have the `step` fact available in order to limit the ansible tasks to only run on a specific step. Note that this runs on each step before the deploy steps tasks, the puppet run, and the container deployment.

Typically youll see this used when, to configure a service, you need to execute an Ansible role that has special requirements for the Ansible inventory.

Such is the case for deploying OpenShift on baremetal via TripleO. The Ansible role for deploying OpenShift requires several hosts and groups to exist in the inventory, so we set those up in `external_deploy_tasks`:

```
- name: generate openshift inventory for openshift_master service
  copy:
    dest: "{{playbook_dir}}/openshift/inventory/{{tripleo_role_name}}_
↪openshift_master.yml"
    content: |
      {% if master_nodes | count > 0%}
      masters:
        hosts:
        {% for host in master_nodes %}
        {{host.hostname}}:
            {{host | combine(openshift_master_node_vars) | to_nice_yaml() |␣
↪indent(6)}}
        {% endfor %}
      {% endif %}

      {% if new_masters | count > 0 %}
      new_masters:
        hosts:
        {% for host in new_masters %}
        {{host.hostname}}:
            {{host | combine(openshift_master_node_vars) | to_nice_yaml() |␣
↪indent(6)}}
        {% endfor %}

      new_etcd:
        children:
          new_masters: {}
      {% endif %}

      etcd:
        children:
          masters: {}

      OSEv3:
        children:
          masters: {}
          nodes: {}
          new_masters: {}
          new_nodes: {}
          {% if groups['openshift_glusterfs'] | default([]) %}glusterfs: {}{%␣
↪endif %}
```

In the case of OpenShift, Ansible itself is also called as a command from here, using variables and the inventory thats generated in this section. This way we dont need to mix the inventory that the overcloud deployment itself is using with the inventory that the OpenShift deployment uses.

Deploy steps tasks

These are Ansible tasks that take place in the overcloud nodes. Note that like any other service, these tasks will only execute on the nodes whose role has this service enabled. Youll find this as the `deploy_steps_tasks` section in the service templates. These actions are also ran as part of the deployment steps, so youll have the `step` fact available in order to limit the ansible tasks to only run on a specific step. Note that on each step, this runs after the external deploy tasks, but before the puppet run and the container deployment.

Typically youll run quite simple tasks in this section, such as setting the boot parameters for the nodes. Although, you can also run more complex roles, such as the IPSec service deployment for TripleO:

```
- name: IPSEC configuration on step 1
  when: step == '1'
  block:
  - include_role:
      name: tripleo-ipsec
    vars:
      map_merge:
      - ipsec_configure_vips: false
        ipsec_skip_firewall_rules: false
      - {get_param: IpsecVars}
```

This type of deployment applies for services that are better tied to TripleOs Ansible inventory or that dont require a specific inventory to run.

Container-related parameters

This covers the sections that allow you to write a containerized service for TripleO.

Containerized services brought a big change to TripleO. From packaging puppet manifests and relying on them for configuration, we now have to package containers, make sure the configuration ends up in the container somehow, then run the containers. Here I wont describe the whole workflow of how we containerized OpenStack services, but instead Ill describe what you need to know to deploy a containerized service with TripleO.

`puppet_config` section

Before getting into the deployment steps where TripleO starts running services and containers, there is a step where puppet is ran in containers and all the needed configurations are created. The `puppet_config` section controls this step.

There are several options we can pass here:

- `puppet_tags`: This describes the puppet resources that will be allowed to run in puppet when generating the configuration files. Note that deeper knowledge of your manifests and what runs in puppet is required for this. Else, it might be better to generate the configuration files with Ansible with the mechanisms described in previous sections of this document. Any service that specifies tags will have the default tags of `'file,concat,file_line,augeas,cron'` appended to the setting. To know what settings to set here, as mentioned, you need to know your puppet manifests.

But, for instance, for keystone, an appropriate setting would be: `keystone_config`. For our etcd example, no tags are needed, since the default tags we set here are enough.

- `config_volume`: The name of the directory where configuration files will be generated for this service. Youll eventually use this to know what location to bind-mount into the container to get the configuration. So, the configuration will be persisted in: `/var/lib/config-data/puppet-generated/<config_volume>`
- `config_image`: The name of the container image that will be used for generating configuration files. This is often the same container that the runtime service uses. Some services share a common set of config files which are generated in a common base container. Typically youll get this from a parameter you pass to the template, e.g. `<Service name>Image` or `<Service name>ConfigImage`. Dealing with these images requires dealing with the container image prepare workflow. The parameter should point to the specific image to be used, and itll be pulled from the registry as part of the deployment.
- `step_config`: Similarly to the `step_config` thats described earlier in this document, this setting controls the puppet manifest that is ran for this service. The aforementioned puppet tags are used along with this manifest to generate a config directory for this container.

One important thing to note is that, if youre creating a containerized service, you dont need to output a `step_config` section from the `roles_data` output. TripleO figured out if youre creating a containerized service by checking for the existence of the `docker_config` section in the `roles_data` output.

`kolla_config` section

As you might know, TripleO uses kolla to build the container images. Kolla, however, not only provides the container definitions, but provides a rich framework to extend and configure your containers. Part of this is the fact that it provides an entry point that receives a configuration file, with which you can modify several things from the container on start-up. We take advantage of this in TripleO, and its exactly what the `kolla_config` represents.

For each container we create, we have a relevant `kolla_config` entry, with a mapping key that has the following format:

```
/var/lib/kolla/config_files/<container name>.json
```

This, contains YAML that represents how to map config files into the container. In the container, this typically ends up mapped as `/var/lib/kolla/config_files/config.json` which kolla will end up reading.

The typical configuration settings we use with this setting are the following:

- `command`: This defines the command well be running on the container. Typically itll be the command that runs the server. So, in the example you see `/usr/bin/etcd ...`, which will be the main process running.
- `config_files`: This tells kolla where to read the configuration files from, and where to persist them to. Typically what this is used for is that the configuration generated by puppet is read from the host as read-only, and mounted on `/var/lib/kolla/config_files/src`. Subsequently, it is copied on to the right location by the kolla mechanisms. This way we make sure that the container has the right permissions for the right user, given well typically be in another user namespace in the container.

- `permissions`: As you would expect, this sets up the appropriate permissions for a file or set of files in the container.

`docker_config` section

This is the section where we tell TripleO what containers to start. Here, we explicitly write on which step to start which container. Steps are set as keys with the `step_<step number>` format. Inside these, we should set up keys with the specific container names. In our example, were running only the etcd container, so we use a key called `etcd` to give it such a name. Paunch or tripleo_container_manage Ansible role will read these parameters, and start the containers with those settings.

Heres an example of the container definition:

```
step_2:
  etcd:
    image: {get_param: ContainerEtcdImage}
    net: host
    privileged: false
    restart: always
    healthcheck:
      test: /openstack/healthcheck
    volumes:
      - /var/lib/etcd:/var/lib/etcd
      - /etc/localtime:/etc/localtime:ro
      - /var/lib/kolla/config_files/etcd.json:/var/lib/kolla/config_files/
↪config.json:ro
      - /var/lib/config-data/puppet-generated/etcd/:/var/lib/kolla/config_
↪files/src:ro
    environment:
      - KOLLA_CONFIG_STRATEGY=COPY_ALWAYS
```

This is what were telling TripleO to do:

- Start the container on step 2
- Use the container image coming from the `ContainerEtcdImage` heat parameter.
- For the container, use the hosts network.
- The container is not privileged.
- The container will use the `/openstack/healthcheck` endpoint for healthchecking
- We tell it what volumes to mount
  - Aside from the necessary mounts, note that were bind-mounting the file `/var/lib/kolla/config_files/etcd.json` on to `/var/lib/kolla/config_files/config.json`. This will be read by kolla in order for the container to execute the actions we configured in the `kolla_config` section.
  - We also bind-mount `/var/lib/config-data/puppet-generated/etcd/`, which is where the puppet ran (which was ran inside a container) persisted the needed configuration files. We bind-mounted this to `/var/lib/kolla/config_files/src` since we told kolla to copy this to the correct location inside the container on the `config_files` section thats part of `kolla_config`.

- Environment tells the container engine which environment variables to set
  - We set `KOLLA_CONFIG_STRATEGY=COPY_ALWAYS` in the example, since this tells kolla to always execute the `config_files` and `permissions` directives as part of the kolla entry point. If we dont set this, it will only be executed the first time we run the container.

`container_puppet_tasks` section

These are containerized puppet executions that are meant as bootstrapping tasks. They typically run on a bootstrap node, meaning, they only run on one relevant node in the cluster. And are meant for actions that you should only execute once. Examples of this are: creating keystone endpoints, creating keystone domains, creating the database users, etc.

The format for this is quite similar to the one described in `puppet_config` section, except for the fact that you can set several of these, and they also run as part of the steps (you can specify several of these, divided by the `step_<step number>` keys).

Note: This was docker_puppet_tasks prior to the Train cycle.

1.2.2 Release Management

Releases Overview

Before reading this document and being involved in TripleO release management, its suggested to read the OpenStack Release Management guide.

Most of TripleO projects follows the independent release model. We will be creating stable branches based on our long term supported releases going forward. The details can be found on the releases repository.

All information about previous releases can be found on https://releases.openstack.org. This page will document the process of releasing TripleO projects.

The tagging convention can be discussed with the PTL or the Release Liaison of TripleO.

For puppet-tripleo, we also need to update metadata.json file:

```
"version": "X.Y.Z",
```

For other projects, there is no need to update anything since the release will be ready by pbr.

Note: Puppet OpenStack modules release management is documented here: https://docs.openstack.org/puppet-openstack-guide/releases.html#how-to-release-puppet-modules

Once this is done, you can submit a patch in openstack/releases and per project to modify the YAML. The openstack/releases project provides tooling to update these files. See the new-release command. You can also update the yaml files manually as necessary. Example with tripleo-heat-templates, edit deliverables/pike/tripleo-heat-templates.yaml:

```
---
launchpad: tripleo
release-type: python-pypi
team: tripleo
type: other
repository-settings:
  openstack/tripleo-heat-templates: {}
releases:
  - version: 15.0.0
    projects:
      - repo: openstack/tripleo-heat-templates
        hash: 1ffbc6cf70c8f79cb3a1e251c9b1e366843ab97c
  - version: 15.1.0
    projects:
      - repo: openstack/tripleo-heat-templates
        hash: ec8955c26a15f3c9e659b7ae08223c544820af03
  - version: 16.0.0
    projects:
      - repo: openstack/tripleo-heat-template
        hash: <MY NEW HASH>
```

Once the file is edited, you can submit it and OpenStack release team will review it. Note that the patch requires +1 from TripleO PTL or TripleO Release Liaison.

The process of branching is also done by Release tools, and you need to change the YAML to specify where we want to branch. Example with tripleo-heat-templates, edit deliverables/ocata/tripleo-heat-templates.yaml:

```
---
launchpad: tripleo
release-type: python-pypi
team: tripleo
type: other
repository-settings:
  openstack/tripleo-heat-templates: {}
branches:
  - name: stable/xena
    location: 16.0.0
releases:
  - version: 15.0.0
    projects:
      - repo: openstack/tripleo-heat-templates
        hash: 1ffbc6cf70c8f79cb3a1e251c9b1e366843ab97c
  - version: 15.1.0
    projects:
      - repo: openstack/tripleo-heat-templates
        hash: ec8955c26a15f3c9e659b7ae08223c544820af03
  - version: 16.0.0
    projects:
      - repo: openstack/tripleo-heat-template
        hash: <MY NEW HASH>
```

Keep in mind that tags, branches, release notes, announcements are generated by the tooling and nothing has to be done manually, except what is documented here.

Releases for RDO

Due to TripleOs switch to the independent model, the TripleO project needs to cut tags at the end of cycles that will not be supported in the long term. These tags are used by the RDO release process to include a build of the TripleO rpms in the RDO release. The process to create the intermediate release would be as follows.

Update required metadata

Some projects like puppet-tripleo and puppet-pacemaker require the metadata be updated in the repository prior to cutting a tag. If the metadata is not updated, the tagging patch to openstack/releases will fail CI.

For puppet-tripleo and puppet-pacemaker, update the version information to represent the next tag version (e.g. 16.1.0).

Get latest promoted content

After the previous metadata updates are available in the latest promoted content, fetch the version information from RDO which contains the git repository hashes.

An example where this could be found is:

```
https://trunk.rdoproject.org/centos8-master/current-tripleo/versions.csv
```

Note: You will needed to adjust the centos8 to centos9 as necessary.

Prepare version tags

Based on the versions.csv data, an openstack/releases patch needs to be created to tag the release with the provided hashes. You can determine which TripleO projects are needed by finding the projects tagged with team: tripleo. An example review. Please be aware of changes between versions and create the appropriate version number as necessary (e.g. major, feature, or bugfix).

Note: If this is a long term release, this patch should include a stable branch.

Notify RDO team of tags

Once the release has been created, make sure the RDO team not has been notified of the new tags. They will update the RDO release items to ensure that the given openstack release will contained the pinned content.

1.2.3 Primer python-tripleoclient and tripleo-common

This document gives an overview of how python-tripleoclient provides the cli interface for TripleO. In particular it focuses on two key aspects of TripleO commands: where they are defined and how they (very basically) work.

Whilst python-tripleoclient provides the CLI for TripleO, it is in tripleo-common that the logic behind a given command resides. So interfacing with OpenStack services such as Heat, Nova or Mistral typically happens in tripleo-common.

For this primer we will use a specific example command but the same applies to any TripleO cli command to be found in the TripleO documentation or in any local deployment (or even in TripleO CI) logfiles.

The example used here is:

```
openstack overcloud container image build
```

This command is used to build the container images listed in the tripleo-common file overcloud_containers.yaml using Kolla.

See the Building Containers Deploy Guide for more information on how to use this command as an operator.

One of the TripleO CI jobs that executes this command is the tripleo-build-containers-centos-7 job. This job invokes the overcloud container image build command in the build.sh.j2 template:

```
openstack overcloud container image build \
--config-file $TRIPLEO_COMMON_PATH/container-images/overcloud_containers.yaml␣
↪\
--kolla-config-file {{ workspace }}/kolla-build.conf \
```

The relevance of showing this is simply to serve as an example in the following sections. First we see how to identify *where* in the tripleoclient code a given command is defined, and then *how* the command works, highlighting a recurring pattern common to all TripleO commands.

TripleO commands: *where*

Luckily the location of all TripleO commands is given in the list of `entry_points` in the python-tripleoclient setup.cfg file. Each *key*=*value* pair has a key derived from the TripleO command. Taking the command, omit the initial *openstack* and link subcommands with underscore instead of whitespace. That is, for the **openstack overcloud container image build** command the equivalent entry is **overcloud_container_image_build**:

```
[entry_points]
openstack.cli.extension =
    tripleoclient = tripleoclient.plugin
```

(continues on next page)

(continued from previous page)

```
openstack.tripleoclient.v1 =
...
    overcloud_container_image_build = tripleoclient.v1.container_
↪image:BuildImage
```

The value in each *key=value* pair provides us with the file and class name used in the tripleoclient namespace for this command. For **overcloud_container_image_build** we have **tripleoclient.v1.container_image:BuildImage**, which means this command is defined in a class called **BuildImage** inside the tripleoclient/v1/container_image.py file.

TripleO commands: *how*

Obviously each TripleO command works differently in that they are doing different things - deploy vs upgrade the undercloud vs overcloud etc. However there **is** at least one commonality which we highlight in this section. Each TripleO command class defines a get_parser function and a take_action function.

The get_parser is where all command line arguments are defined and take_action is where tripleo-common is invoked to perform the task at hand, building container images in this case.

Looking inside the **BuildImage** class we find:

```
def get_parser(self, prog_name):
...
    parser.add_argument(
        "--config-file",
        dest="config_files",
        metavar='<yaml config file>',
        default=[],
        action="append",
        help=_("YAML config file specifying the images to build. May be "
               "specified multiple times. Order is preserved, and later "
               "files will override some options in previous files. "
               "Other options will append. If not specified, the default "
               "set of containers will be built."),
    )
    parser.add_argument(
        "--kolla-config-file",
```

Here we can see where the two arguments shown in the introduction above are defined: **config-file** and **kolla-config-file**. You can see the default values and all other attributes for each of the command parameters there.

Finally we can look for the take_action function to learn more about how the command actually works. Typically the take_action function will have some validation of the provided arguments before calling out to tripleo-common to actually do the work (build container images in this case):

```
from tripleo_common.image import kolla_builder
...
def take_action(self, parsed_args):
...
```

(continues on next page)

(continued from previous page)

```
    try:
        builder = kolla_builder.KollaImageBuilder(parsed_args.config_files)
        result = builder.build_images(kolla_config_files,
```

Here we can see the actual image build is done by the **kolla_builder.KollaImageBuilder** class **build_images** function. Looking in tripleo-common we can follow that python namespace to find the definition of **build_images** in the tripleo_common/image/kolla_builder.py file:

```
def build_images(self, kolla_config_files=None, excludes=[],
                 template_only=False, kolla_tmp_dir=None):
    cmd = ['kolla-build']
...
```

1.2.4 Upgrades Development

This section is intended to give a better understanding of the upgrade/update process in TripleO. As well as a walkthrough for developers on the way upgrade workflow enables OpenStack services upgrade.

Overcloud Major Upgrade Workflow and CLI

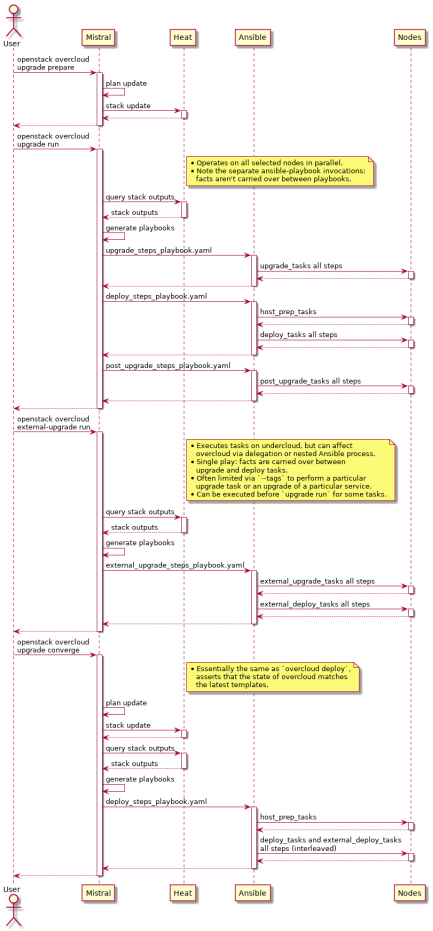
The purpose of this documentation is to deep-dive into the code which delivers the major upgrade workflow in TripleO. For information about the steps an operator needs to perform when running this upgrade please see the operator docs.

The major upgrade workflow is delivered almost exclusively via Ansible playbook invocations on the overcloud nodes. Heat is used to generate the Ansible playbooks (during the prepare command at the beginning, and converge command at the end of the upgrade). The Queens_upgrade_spec may be of interest in describing the design of the workflow.

CLI code is in python-tripleoclient, mistral workflows and actions in tripleo-common, and upgrade tasks in tripleo-heat-templates. The following sections dive into the details top-down per individual CLI commands which are used to deliver the major upgrade:

- *openstack overcloud upgrade prepare $ARGS*
- *openstack overcloud upgrade run $ARGS*
- *openstack overcloud external-upgrade run $ARGS*
- *openstack overcloud upgrade converge $ARGS*

You might also find it helpful to consult this high-level diagram as you read the following sections:



openstack overcloud upgrade prepare $ARGS

The entry point for the upgrade CLI commands, *prepare*, *run* and *converge*, is given in the python-tripleoclient setup.cfg. All three are also defined in the same file, overcloud-upgrade.py.

The prepare Heat stack update does not apply any TripleO configuration and is exclusively used to generate the Ansible playbooks that are subsequently invoked to deliver the upgrade.

As you can see the UpgradePrepare class inherits from DeployOvercloud. The reason for this is to prevent duplication of the logic concerned with validating the configuration passed into the prepare command (all the -e env.yaml files), as well as updating_the_swift_stored_plan with the overcloud configuration.

The prepare_env_file is automatically prepended to the list of environment files passed to Heat (as specified by prepare_command_prepends). It contains resource_registry and parameter_defaults which are intended to be in effect during the upgrade.

As a result the UpgradePrepare class inherits all the Deploy_parser_arguments, including `--stack` and `-e` for the additional environment files. We explicitly set the update_plan_only argument so that the Heat stack update does not get executed by the parent class and returns after completing all the template processing.

Instead, the Heat stack update is performed by a mistral workflow. On the client side the hook is in the update method defined in package_update.py. This invokes the package_update_plan mistral workflow in tripleo-common. The package_update_plan workflow has a number of tasks, one of which invokes the heat stack update using the update_stack_action.

Back on the tripleoclient side, we use base_wait_for_messages to listen for messages on the Zaqar_queue that is used by the mistral workflow.

The operator must include all environment files previously used with the *overcloud deploy* command. It is especially important that the operator includes the environment file containing the references for the target version container images.

See the operator docs for pointers to how that file is generated and for reference it will look something like

```
parameter_defaults:
  DockerAodhApiImage: 192.168.24.1:8787/queens/centos-binary-aodh-
↪api:current-tripleo-rdo
  DockerAodhConfigImage: 192.168.24.1:8787/queens/centos-binary-aodh-
↪api:current-tripleo-rdo
  DockerAodhEvaluatorImage: 192.168.24.1:8787/queens/centos-binary-
↪aodh-evaluator:current-tripleo-rdo
  DockerAodhListenerImage: 192.168.24.1:8787/queens/centos-binary-
↪aodh-listener:current-tripleo-rdo
```

Once the Heat stack update has been completed successfully and the stack is in UPDATE_COMPLETE state, you can download the configuration ansible playbooks using the config download cli

```
[stack@521-m--undercloud ~]$ source stackrc
(undercloud) [stack@521-m--undercloud ~]$ openstack overcloud config␣
↪download --config-dir MYCONFIGDIR
The TripleO configuration has been successfully generated into:␣
↪MYCONFIGDIR/tripleo-gep7gh-config
```

and you can inspect the ansible playbooks which are used by the *upgrade run* before executing them.

openstack overcloud upgrade run $ARGS

Unlike the first step in the workflow, the *upgrade prepare*, the UpgradeRun class does not inherit from DeployOvercloud. There is no need for the operator to pass all the environment files and configuration here. The template processing and update of the stack and swift stored plan have already taken place. The ansible playbooks are ready to be retrieved by config download as demonstrated above. The upgrade run operation thus will simply execute those ansible playbooks generated by the upgrade prepare command, against the nodes specified in the parameters.

Either `--nodes` or `--roles` parameters are used to limit the ansible playbook execution to specific nodes. Both `--roles` and `--nodes` are used by ansible with the tripleo-ansible-inventory. This creates the ansible inventory based on the Heat stack outputs, so that for example `Controller` and `overcloud-controller-0` are both valid values for the ansible-playbook `--limit` parameter.

See overcloud upgrade run for additional information.

As documented in the major upgrade documentation and the nodes_or_roles_helptext, the operator *must* use `--roles` for the controllers. Upgrading the controlplane, one node at a time is currently not supported, mainly due to limitations in the pacemaker cluster upgrade which needs to occur across all nodes in the same operation. The operator may use `--roles` for non controlplane nodes or may prefer to specify one or more specific nodes by name with `--nodes`. In either case the value specified by the operator is simply passed through to ansible as the limit_hosts parameter.

The `--ssh-user` and all other parameters are similarly collected and passed to the ansible invocation which starts on the client side in the run_update_ansible_action method call. The `--skip-tags` parameter can be used to skip certain ansible tasks with the ansible-skip-tags ansible-playbook parameter. The allowed `--skip-tags` values are restricted to a predefined set, validated against MAJOR_UPGRADE_SKIP_TAGS. Finally, the `--playbook` parameter as the name suggests is used to specify the ansible playbook(s) to run. By default and as you can see in the definition, this defaults to a special value all which causes all-upgrade-playbooks-to-run. The value of all_playbooks in that previous reference, is stored in the MAJOR_UPGRADE_PLAYBOOKS constant.

As with the *upgrade prepare*, for *upgrade run* a mistral workflow is used to perform the main operation, which in this case is execution of the ansible playbooks. On the client side the update_nodes_workflow_invocation is where mistral is invoked and takes as workflow input the various collected parameters described above. You can see that the update_nodes_workflow which lives in tripleo-common has parameters defined under the input: section which correspond to the *openstack overcloud upgrade run* parameters.

There are two main tasks in the update_nodes_workflow, the download-config_action which is invoked in a first download_config task, and the ansible-playbook_action action which is invoked in the node_update task. This is ultimately where ansible-playbook-is-executed with processutils.execute.

Finally back on the client side we listen for messages on the run_zaqar_queue before declaring the upgrade-run-success!

openstack overcloud external-upgrade run $ARGS

The *external-upgrade run* command is used to upgrade the services whose deployment (and upgrade) procedure is not tied to execution on particular overcloud nodes. The deployment/upgrade procedures are thus executed from the undercloud, even though a full overcloud inventory is available for use.

The *external upgrade playbook* first executes *external_upgrade_tasks* and then *external_deploy_tasks*. The execution happens within the same Ansible play, so facts from *external_upgrade_tasks* are carried over to *external_deploy_tasks*. This is a mechanism which will allow you to amend what your deploy tasks do based on whether an upgrade is being run or not.

Often its not desirable to run the tasks for all services at the same time, so *external-upgrade run* supports `--tags` argument to limit which tasks are run.

The mechanisms of *external-upgrade* and *external-update* commands and Ansible tasks are the same, but two commands and task hooks are provided because generally in OpenStack we distinguish minor update vs. major upgrade workflows. If your service only has one type of upgrade, you can make the *external_update_tasks* the same as *external_upgrade_tasks* by using YAML anchors and references.

openstack overcloud upgrade converge $ARGS

The UpgradeConverge class like the UpgradePrepare class also inherits from the DeployOvercloud class thus getting all of its parameters and template processing. The operator needs to pass in all Heat environment files used as part of the upgrade prepare including the container images file.

The main objective of the upgrade converge operation is to unset the upgrade specific parameters that have been set on the overcloud Heat stack as part of prepare. These are unset using the converge_env_file which is included in the list of client_converge_env_files passed to the Heat stack update.

The converge applies all TripleO configuration against all overcloud nodes and thus serves as a sanity check that the overcloud was successfully upgraded, since the same configuration will already have been applied. The converge will also leave the Heat stack in a good state for subsequent updates, for instance scaling to add nodes.

As these values are set in parameter_defaults a Heat stack update is required against the overcloud Heat stack to explicitly unset them. In particular and as pointed out in the operator_converge_docs until converge has completed, any operations that require a Heat stack update will likely fail, as the noop of the DeploymentSteps in the prepare_env_file in particular means none of the usual docker/puppet/* config is applied. Setting something with parameter_defaults means it is used until explicitly unset via parameter_defaults as that value will override any other default value specified via the tripleo-heat-templates.

Unlike the prepare command there is no mistral workflow here and instead we rely on the parent DeployOvercloud class to invoke the converge_heat_stack_update and so the implementation is also simpler.

Upgrade CLI developer workflow

This section will give some examples of a potential developer workflow for testing fixes or in-progress gerrit reviews against python-tripleoclient, tripleo-common or tripleo-heat-templates for the upgrade workflow. This may be useful if you are working on an upgrades related bug for example.

Making changes to the ansible playbooks

If there is a failure running one of the upgrades related ansible playbooks, you might need to examine and if necessary fix the related ansible task. The tasks themselves live in each of the tripleo-heat-templates service manifests, under the upgrade_tasks section of the template outputs. For example see the containerized rabbitmq_upgrade_tasks.

If you make a change in service upgrade_tasks, then to test it you will need to

1. Patch the tripleo-heat-templates in your environment with the fix
2. Rerun *openstack overcloud upgrade prepare $ARGS*, so that the resulting ansible playbooks include your fix.
3. Finally run the playbooks with *openstack overcloud upgrade run $ARGS*.

Assuming you are using the default /usr/share/openstack-tripleo-heat-templates directory for the deployment templates you can use the following as just one example:

```
# backup tht in case you want to revert - or just yum re-install ;)
sudo cp -r /usr/share/openstack-tripleo-heat-templates \
    /usr/share/openstack-tripleo-heat-templates.ORIG
# Apply patch from gerrit e.g. https://review.opendev.org/#/c/563073/
curl -4sSL 'https://review.opendev.org/changes/563073/revisions/
↪current/patch?download' | \
    base64 -d | \
    sudo patch -d /usr/share/openstack-tripleo-heat-templates/ -p1
```

Making changes to the upgrades workflow

If instead you need to add or fix something in the upgrades workflow itself, for example to handle a new parameter needed passed through to ansible, or any other change, you will need to patch python-tripleoclient and tripleo-common, depending on whether your fixes extend to the mistral workflow too.

There are many ways to patch your environment and the following is a different approach to the one used in the tripleo-heat-templates above where we patched the installed templates in place. In the following examples instead we clone tripleo-common and tripleoclient, patch them using gerrit reviews and then re-install from source.

> **Note:** The following example commands include complete removal and replacement of the installed tripleoclient and tripleo-common!

Patching python-tripleoclient:

```
# python-tripleoclient - clone source, patch from gerrit and install
git clone https://github.com/openstack/python-tripleoclient.git -b␣
↪stable/queens ~/python-tripleoclient
pushd ~/python-tripleoclient

# Apply patches from gerrit e.g. https://review.opendev.org/#/c/
↪564267
curl "https://review.opendev.org/changes/564267/revisions/current/
↪patch" | \
      base64 --decode > /home/stack/"564267.patch"
patch -N -p1 -b -z .first < /home/stack/564267.patch
# Remove current version and re-install
sudo rm -rf /usr/lib/python2.7/site-packages/python_tripleoclient*
sudo rm -rf /usr/lib/python2.7/site-packages/tripleoclient
sudo python setup.py clean --all install
popd
```

Patching tripleo-common:

> **Note:** After switching to containerized undercloud, local tripleo-common changes to be applied in all Mistral containers.

```
# tripleo-common - clone from source, patch from gerrit and install
git clone https://github.com/openstack/tripleo-common -b stable/
↪queens
pushd ~/tripleo-common
# Apply patches from gerrit e.g. https://review.opendev.org/#/c/
↪562995
curl "https://review.opendev.org/changes/562995/revisions/current/
↪patch" | \
     base64 --decode > /home/stack/"562995.patch"
patch -N -p1 -b -z .first < /home/stack/562995.patch
# Remove current version and re install
```

(continues on next page)

(continued from previous page)

```
sudo rm -rf /usr/lib/python2.7/site-packages/tripleo_common*
sudo python setup.py clean --all install
popd
sudo cp /usr/share/tripleo-common/sudoers /etc/sudoers.d/tripleo-
↪common
```

Finally you need to update the mistral workbooks with the newly installed versions. In code block above, the tripleo-common change at 562995 has changed package_update.yaml and so that is what we need to update here:

```
mistral workbook-update /usr/share/tripleo-common/workbooks/package_
↪update.yaml
# Since entry_points.txt is affected next steps are required:
# Re populate mistral db and restart services
sudo mistral-db-manage  populate
sudo systemctl restart openstack-mistral-api.service
sudo systemctl restart openstack-mistral-engine.service
sudo systemctl restart openstack-mistral-executor.service
```

Minor version update

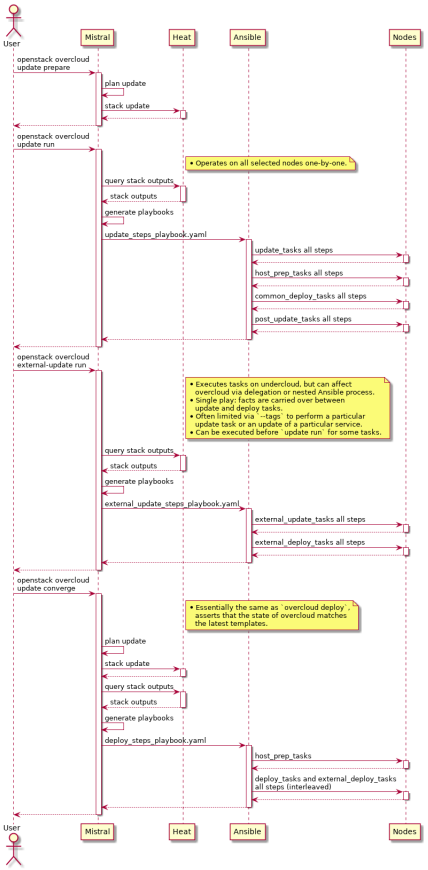
Assuming operator-level familiarity with the minor updates, lets look at individual pieces in more detail.

How update commands work

The following subsections describe the individual update commands:

- *openstack overcloud update prepare*
- *openstack overcloud update run*
- *openstack overcloud external-update run*
- *openstack overcloud update converge*

You might also find it helpful to consult this high-level diagram as you read:



openstack overcloud update prepare

The *update prepare* command performs a Heat stack update, mapping some resources to `OS::Heat::None` in order to prevent the usual deployment config management tasks being performed (running Puppet, starting containers, running external installers like ceph-ansible). See the update prepare environment file.

The purpose of this stack update is to regenerate fresh outputs of the Heat stack. These outputs contain Ansible playbooks and task lists which are then used in the later in the *update run* phase.

openstack overcloud update run

The *update run* command utilizes the previously generated Heat stack outputs. It downloads the playbook yamls and their included task list yaml via the config-download mechanisms, and executes the update steps playbook.

The command accepts `--nodes` or `--roles` argument to limit which nodes will be targeted during a particular *update run* execution. Even if the limit matches multiple nodes (e.g. all nodes within one role), the play is executed with `serial: 1`, meaning that all actions are finished on one node before starting the update on another.

The play first executes *update_steps_tasks.yaml* which are tasks collected from the `update_tasks` entry in composable services.

After the update tasks are finished, deployment workflow is performed on the node being updated. That means reusing *host_prep_tasks.yaml* and *common_deploy_steps_tasks.yaml*, which are executed like on

a fresh deployment, except during minor update theyre within a play with the aforementioned `serial: 1` limiting.

Finally, `post_update_tasks` are executed. They are utilized by services which need to perform something *after* deployment workflow during the minor update. The update of the node is complete and the Ansible play continues to update another node.

openstack overcloud external-update run

The *external-update run* command is used to update the services whose deployment (and update) procedure is not tied to execution on particular overcloud nodes. The deployment/update procedures are thus executed from the undercloud, even though a full overcloud inventory is available for use.

The *external update playbook* first executes *external_update_tasks* and then *external_deploy_tasks*. The execution happens within the same Ansible play, so facts from *external_update_tasks* are carried over to *external_deploy_tasks*. This is a mechanism which will allow you to amend what your deploy tasks do based on whether an update is being run or not.

Often its not desirable to run the tasks for all services at the same time, so *external-update run* supports `--tags` argument to limit which tasks are run.

The mechanisms of *external-upgrade* and *external-update* commands and Ansible tasks are the same, but two commands and task hooks are provided because generally in OpenStack we distinguish minor update vs. major upgrade workflows. If your service only has one type of upgrade, you can make the *external_update_tasks* the same as *external_upgrade_tasks* by using YAML anchors and references.

openstack overcloud update converge

Note: Update Converge is only required for versions less than Wallaby. Update Converge has been removed for Wallaby and beyond.

The *update converge* command performs a Heat stack update, reverting the previous `OS::Heat::None` resource mappings back to the values used for regular deployments and configuration updates, and potentially also resets some parameter values. For environments with Ceph, majority of this already happened on *ceph-upgrade run*, so the final *update converge* effectively just resets the CephAnsiblePlaybook parameter.

See the update converge environment file.

The purpose of this stack update is to re-run config management mechanisms and assert that the overcloud state matches what is provided by the templates and environment files.

Writing update logic for a service

Simple config/image replacement

If the service is managed by Paunch or tripleo_container_manage Ansible role, it may be that theres no need to write any update tasks. Paunch or tripleo_container_manage can automatically handle simple updates: change in configuration or change of container image URL triggers automatic removal of the old container and creation of new one with latest config and latest image. If thats all the service needs for updates, you dont need to create any `update_tasks`.

Custom tasks during updates

If the service is not managed by Paunch nor tripleo_container_manage, or if the simple container replacement done by Paunch is not sufficient for the service update, you will need to include custom update logic. This is done via providing these outputs in your composable service template:

- `update_tasks` these are executed before deployment tasks on the node being updated.
- `post_update_tasks` these are executed after deployment tasks on the node being updated.

Update tasks are generally meant to bring the service into a stopped state (sometimes with pre-fetched new images, this is necessary for services managed by Pacemaker). Then the same workflow as during deployment is used to bring the node back up into a running state, and the post-update tasks can then perform any actions needed after the deployment workflow.

Similarly as deployment tasks, the update tasks and post-update tasks are executed in steps.

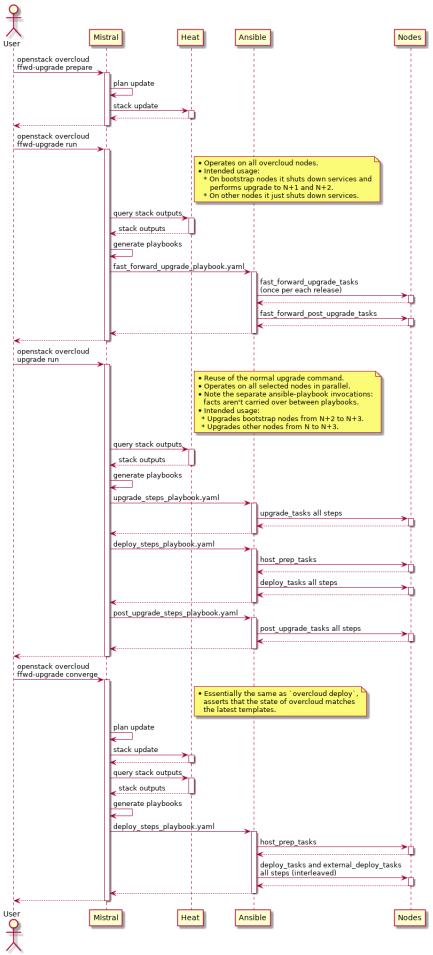
TripleO Fast Forward Upgrade (FFU) N -> N+3

For a detailed background on how the Fast Forward Upgrade (FFU) workflow was proposed please refer to the relevant spec.

For a guide on running the FFU in your environment see the FFU Deploy Guide.

This document will explore some of the technical details of the Newton to Queens FFU specifically.

You might find it helpful to consult this high-level diagram as you read on:



At a high level the FFU workflow consists of the following steps:

1. Perform a Minor update on the environment (both undercloud and overcloud) to bring it to the latest Newton. This will include OS level updates, including kernel and openvswitch. As usual for minor update the operator will reboot each node as necessary and so doing this first means the FFU workflow doesnt (also) have to deal with node reboots later on in the process.

2. Perform 3 consecutive major upgrades of the undercloud to bring it to Queens. The undercloud will crucially then have the target version of the tripleo-heat-templates including the fast_forward_upgrade_tasks that will deliver the next stages of the workflow.

3. Generate and then run the fast_forward_upgrade_playbook on the overcloud. This will:

   3.1 First bring down the controlplane services on **all nodes**.

   3.2 Then update packages, migrate databases and any other version specific tasks from Newton to Ocata then Ocata to Pike. This happens only on a **single node of each role**.

4. Finally run the Pike to Queens upgrade on all nodes including the Queens upgrade tasks and service configurations.

Step 3 above is started by first performing a Heat stack update using the Queens tripleo-heat-templates from the Queens upgraded undercloud, but without applying any configuration. This stack update is only used to collect the fast_forward_upgrade_tasks (ffu_tasks) from each of the services deployed in the given environment and generate a fast_forward_upgrade_playbook ansible playbook. This playbook is then executed to deliver steps 3.1 and 3.2 above. See below for more information about how the ffu_tasks are compiled into the fast_forward_upgrade_playbook.

A notable exception worthy of mention is the configuration of Ceph services which is managed by ceph-ansible. That is, for Ceph services there is no collection of fast_forward_upgrade_tasks from the ceph related service manifests in the tripleo-heat-templates and so Ceph is not managed by the generated fast_forward_upgrade_playbook. Instead ceph-ansible will be invoked by the Queens deployment and service configuration in step 4 above.

The Heat stack update performed at the start of step 3 also generates the Queens upgrade_steps_playbook and deploy_steps_playbook ansible playbooks. One notable exception is the configuration of Ceph services which is managed by ceph-ansible Step 4 above (Pike to Queens upgrade tasks and Queens services configuration) is delivered through execution of these Heat stack update generated playbooks. Ceph related upgrade and deployment will be applied here with calls to ceph-ansible.

Amongst other things, the P..Q upgrade_tasks stop and disable those systemd services that are being migrated to run in containers. The Queens deploy steps playbook will then apply the required puppet and docker configuration to start the containers for those services. For this to be possible the Heat stack update which starts step 3 and that generates the ansible playbooks must include the required docker configuration and environment files, including the latest container images and making sure to set the to-be containerized services to refer to the equivalent docker templates for the Heat resource registry.

FFU and tripleo-heat-templates

This section will present an overview of how the fast_forward_upgrade_playbook.yaml is generated from the tripleo-heat-templates.

FFU uses *fast_forward_upgrade_tasks* (ffu_tasks) to define the upgrade workflow. These are normal ansible tasks and they are carried as a list in the outputs section of a given service manifest, see containerized neutron-api for an example.

The ffu_tasks for those services that are enabled in a given deployment are collected in the outputs of the deploy-steps.j2 into a *fast_forward_upgrade_playbook* output. This is then retrieved using the config-download mechanism and written to disk as an ansible playbook.

The *fast_forward_upgrade_tasks* defined for a given service can use the **step** and **release** variables to specify when a given task should be executed. At a high level the fast_forward_upgrade_playbook consists of two loops - there is a very good explanation in /#/c/499221 commit message, but an outer loop for the release (first Ocata tasks then Pike tasks) and then an inner loop for the steps within each release.

The *ffu_tasks* which are set to run in steps 0 to 3 are designated the *fast_forward_upgrade_prep_role_tasks* and these are executed on all nodes for a given role. Then the *ffu_tasks* which have steps 4 to max (currently 9) are designated the *fast_forward_upgrade_bootstrap_role_tasks* and these are only executed on a single node for each role (one controller, one compute etc).

The top level fast_forward_upgrade_playbook.yaml looks like:

```
- hosts: overcloud
  become: true
  tasks:
    - include_tasks: fast_forward_upgrade_release_tasks.yaml
      loop_control:
        loop_var: release
      with_items: {get_param: [FastForwardUpgradeReleases]}
```

The *fast_forward_upgrade_release_tasks.yaml* in turn looks like:

```
- include_tasks: fast_forward_upgrade_prep_tasks.yaml
- include_tasks: fast_forward_upgrade_bootstrap_tasks.yaml
```

The *fast_forward_upgrade_prep_tasks.yaml* specifies the loop with sequence 0 to 3 as explained above:

```
- include_tasks: fast_forward_upgrade_prep_role_tasks.yaml
  with_sequence: start=0 end=3
  loop_control:
  loop_var: step
```

And where the *fast_forward_upgrade_prep_role_tasks.yaml* includes the *ffu_tasks* on all nodes for each role:

```
- include_tasks: Controller/fast_forward_upgrade_tasks.yaml
  when: role_name == 'Controller'
- include_tasks: Compute/fast_forward_upgrade_tasks.yaml
  when: role_name == 'Compute'
...etc
```

Similarly for the *fast_forward_upgrade_bootstrap_tasks.yaml* it specifies the loop sequence for the step variable to be 4 to 9:

```
- include_tasks: fast_forward_upgrade_bootstrap_role_tasks.yaml
  with_sequence: start=4 end=9
  loop_control:
  loop_var: step
```

And where the *fast_forward_upgrade_bootstrap_role_tasks.yaml* include the *ffu_tasks* only on a single node for each role type:

```
- include_tasks: Controller/fast_forward_upgrade_tasks.yaml
  when: role_name == 'Controller' and ansible_hostname == Controller[0]
- include_tasks: Compute/fast_forward_upgrade_tasks.yaml
  when: role_name == 'Compute' and ansible_hostname == Compute[0]
...etc
```

Major upgrades & Minor updates CI coverage

This document tries to give a detailed overview of the current CI coverage for upgrades/updates jobs. Also, it is intended as a guideline to understand how these jobs work, as well as giving some tips for debugging.

Upgrades/Updates CI jobs

At the moment most of the upgrade jobs have been moved from upstream infrastructure to RDO Software Factory job definition due to runtime constraints of the OpenStack infra jobs.

Each of these jobs are defined by a featureset file and a scenario file. The featureset used in a job can be found in the last part of the job type value. This can be found in the ci job definition:

```
- '{trigger}-tripleo-ci-{jobname}-{release}{suffix}':
    jobname: 'centos-7-containers-multinode-upgrades'
    release:
      - pike
      - master
    suffix: ''
    type: 'multinode-1ctlr-featureset011'
    node: upstream-centos-7-2-node
    trigger: gate
```

The scenario used is referenced in the featureset file, in the example above the featureset011 makes use of the following scenarios:

```
composable_scenario: multinode.yaml
upgrade_composable_scenario: multinode-containers.yaml
```

As this job covers the upgrade from one release to another, we need to specify two scenario files. The one used during deployment and the one used when upgrading. Each of these scenario files defines the services deployed in the nodes.

Note: There is a matrix with the different features deployed per feature set here: featureset matrix

---

Currently, two types of upgrade jobs exist:

- multinode-upgrade (mixed-version): In this job, an undercloud with release N+1 is deployed, while the overcloud is deployed with a N release. Execution time is reduced by not upgrading the undercloud , instead the heat templates from the (N+1) undercloud are used when performing the overcloud upgrade.

  > **Note:** If you want your patch to be tested against this job you need to add *RDO Third Party CI* as reviewer or reply with the comment *check-rdo experimental*.

- undercloud-upgrade: This job tests the undercloud upgrade from a major release to another. The undercloud is deployed with release N and upgraded to N+1 release. This job does not deploy an overcloud.

Note: There is an effort to integrate the new tripleo-upgrade role into tripleo-quickstart that defines an unified way to upgrade and update.

Upgrade/Update CI jobs, where to look

The best place to check the current CI jobs status is in the CI Status page. This webpage contains a log of all the TripleO CI jobs, its result status, link to logs, git patch trigger and statistics about the pass/fail rates.

To check the status of the Upgrades/Updates jobs, you need to click the TripleO CI promotion jobs link from CI Status, where you will find the RDO cloud upgrades section:

In this section the CI jobs have a color code, to show its current status in a glance:

```
- Red: CI job constantly failing.
- Yellow: Unstable job, frequent failures.
- Green: CI job passing consistently.
```

If you scroll down after pressing some of the jobs in the section you will find the CI job statistics and the last 100 (or less, it can be edited) job executions. Each of the job executions contains:

```
- Date: Time and date the CI job was triggered
- Length: Job duration
- Reason: CI job result or failure reason.
- Patch: Git ref of the patch tha triggered the job.
- Logs: Link to the logs.
- Branch: Release branch used to run the job.
```

Debugging Upgrade/Update CI jobs

When opening the logs from a CI job it might look a little chaotic (mainly when it is for the first time). Its good to have an idea where you can find the logs you need, so you will be able to identify the cause of a failure or debug some issue.

The first thing to have a look at when debugging a CI job is the console output or full log. When clicking in the job, the following folder structure appears:

```
job-output.json.gz
job-output.txt.gz
logs/
zuul-info/
```

The job execution log is located in the *job-output.txt.gz* file. Once opened, a huge log will appear in front of you. What should you look for?

(1) Find the job result

A good string to search is *PLAY RECAP*. At this point, all the playbooks have been executed and a summary of the runs per node is displayed:

```
PLAY RECAP␣
↪*********************************************************************
127.0.0.2                  : ok=9    changed=0    unreachable=0    ␣
↪failed=0
localhost                  : ok=10   changed=3    unreachable=0    ␣
↪failed=0
subnode-2                  : ok=3    changed=1    unreachable=0    ␣
↪failed=0
undercloud                 : ok=120  changed=78   unreachable=0    ␣
↪failed=1
```

In this case, one of the playbooks executed in the undercloud has failed. To identify which one, we can look for the string **fatal**.:

```
fatal: [undercloud]: FAILED! => {"changed": true, "cmd": "set -o␣
↪pipefail && /home/zuul/overcloud-upgrade.sh 2>&1
| awk '{ print strftime(\"%Y-%m-%d %H:%M:%S |\"), $0; fflush(); }' >␣
↪overcloud_upgrade_console.log",
"delta": "0:00:39.175219", "end": "2017-11-14 16:55:47.124998",
↪"failed": true, "rc": 1,
"start": "2017-11-14 16:55:07.949779", "stderr": "", "stdout": "",
↪"stdout_lines": [], "warnings": []}
```

From this task, we can guess that something went wrong during the overcloud upgrading process. But, where can I find the log *overcloud_upgrade_console.log* referenced in the task?

(2) Undercloud logs

From the *logs directory* , you need to open the *logs/* folder. All undercloud logs are located inside the *undercloud/* folder. Opening it will display the following:

```
etc/      *configuration files*
home/     *job execution logs from the playbooks*
var/      *system/services logs*
```

The log we look for is located in */home/zuul/*. Most of the tasks executed in tripleo-quickstart will store the full script as well as the execution log in this directory. So, this is a good place to have a better understanding of what went wrong.

If the overcloud deployment or upgrade failed, you will also find two log files named:

```
failed_upgrade.log.txt.gz
failed_upgrade_list.log.txt.gz
```

The first one stores the output from the debugging command:

```
openstack stack failures list --long overcloud
```

Which prints out the reason why the deployment or upgrade failed. Although sometimes, this information is not enough to find the root cause for the problem. The *stack failures* can give you a clue of which service is causing the problem, but then youll need to investigate the OpenStack service logs.

(3) Overcloud logs

From the *logs/* folder, you can find a folder named *subnode-2* which contains most of the overcloud logs.:

```
apache/
ceph_conf.txt.gz
deprecations.txt.gz
devstack.journal.gz
df.txt.gz
etc/
home/
iptables.txt.gz
```

(continues on next page)

(continued from previous page)

```
libvirt/
listen53.txt.gz
openvswitch/
pip2-freeze.txt.gz
ps.txt.gz
resolv_conf.txt.gz
rpm-qa.txt.gz
sudoers.d/
var/
```

To access the OpenStack services logs, you need to go to *subnode-2/var/log/* when deploying a baremetal overcloud. If the overcloud is containerized, the service logs are stored under *subnode-2/var/log/containers*.

Replicating CI jobs

Thanks to James Slagle there is now a way to reproduce TripleO CI jobs in any OpenStack cloud. Everything is enabled by the traas project, a set of Heat templates and scripts that reproduce the TripleO CI jobs in the same way they are being run in the Zuul gate.

When cloning the repo, you just need to set some configuration parameters. A set of sample templates have been located under templates/example-environments. The parameters defined in this template are:

```
parameters:
  overcloud_flavor:     [*flavor used for the overcloud instance*]
  overcloud_image:      [*overcloud OS image (available in cloud images)*]
  key_name:             [*private key used to access cloud instances*]
  private_net:          [*network name (it must exist and match)*]
  overcloud_node_count:[*number of overcloud nodes*]
  public_net:           [*public net in CIDR notation*]
  undercloud_image:     [*undercloud OS image (available in cloud images)*]
  undercloud_flavor:    [*flavor used for the undercloud instance*]
  toci_jobtype:         [*CI job type*]
  zuul_changes:         [*List of patches to retrieve*]
```

Note: The CI job type toci_jobtype can be found in the job definition under tripleo-ci/zuul.d.

A good example to deploy a multinode job in RDO Cloud is this sample template. You can test your out patches by appending the refs patch linked with the ^ character:

```
zuul_changes: <project-name>:<branch>:<ref>[^<project-name>:<branch>:<ref>]*
```

This allows you also to test any patch in a local environment without consuming CI resources. Or when you want to debug an environment after a job execution.

Once the template parameters are defined, you just need to create the stack. If we would like to deploy the *rdo-cloud-env-config-download.yaml* sample template we would need to run:

```
cd traas/
openstack stack create traas -t templates/traas.yaml \
    -e templates/traas-resource-registry.yaml \
    -e templates/example-environments/rdo-cloud-env-config-download.yaml
```

This stack will create two instances in your cloud tenant, one for undercloud and another for the overcloud. Once created, the stack will directly call the traas/scripts/traas.sh script which downloads all required repositories to start executing the job.

If you want to follow up the job execution, you can ssh to the undercloud instance and tail the content from the *$HOME/tripleo-root/traas.log*. All the execution will be logged in that file.

CHAPTER

TWO

TRIPLEO ARCHITECTURE

2.1 TripleO Architecture

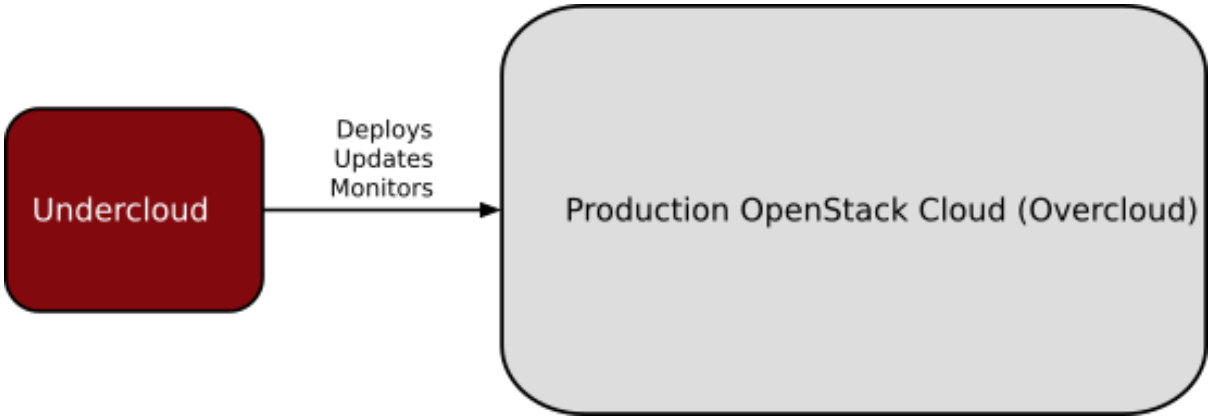
This document lists the main components of TripleO, and gives some description of how each component is used. There are links to additional sources of information throughout the document.

2.1.1 Architecture Overview

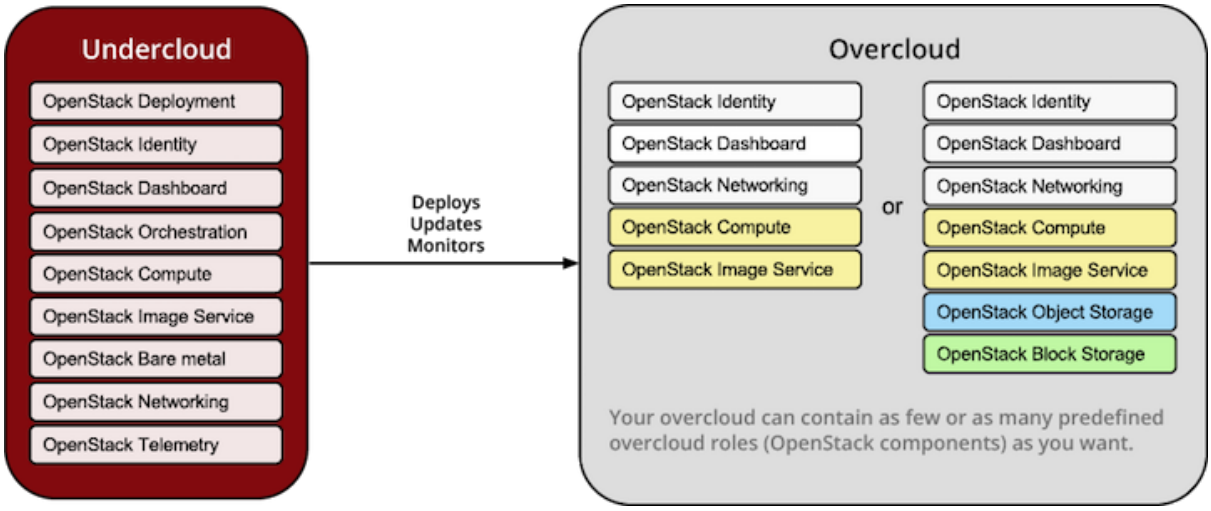
TripleO is a community developed approach and set of tools for deploying, and managing an OpenStack cloud.

TripleO

TripleO is the friendly name for OpenStack on OpenStack. It is an official OpenStack project with the goal of allowing you to deploy and manage a production cloud onto bare metal hardware using a subset of existing OpenStack components.

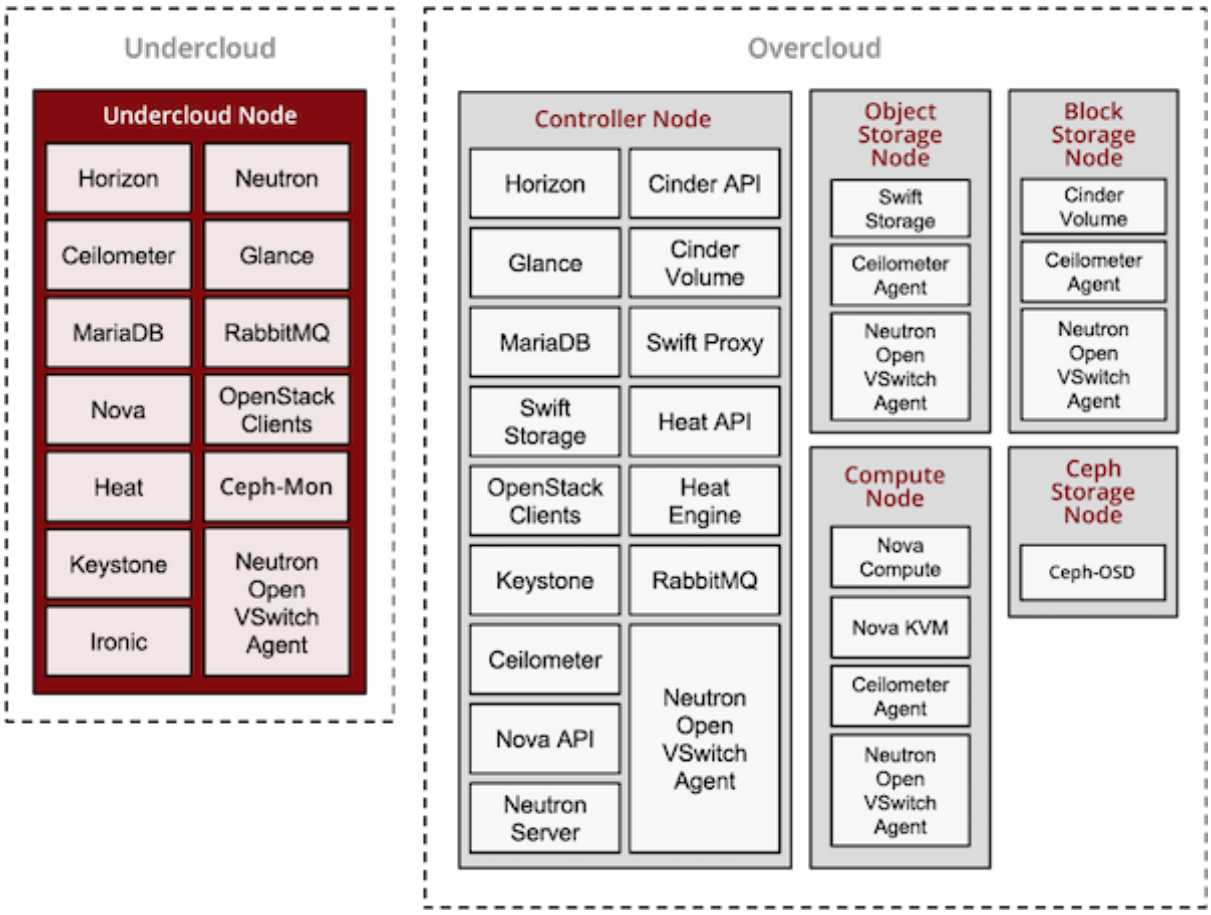


With TripleO, you start by creating an undercloud (a deployment cloud) that will contain the necessary OpenStack components to deploy and manage an overcloud (a workload cloud). The overcloud is the deployed solution and can represent a cloud for any purpose (e.g. production, staging, test, etc).



TripleO leverages several existing core components of OpenStack including Nova, Ironic, Neutron, Heat, Glance and Ceilometer to deploy OpenStack on baremetal hardware. Nova and Ironic are used in the undercloud to manage baremetal instances that comprise the infrastructure for the overcloud. Neutron is utilized to provide a networking environment in which to deploy the overcloud, machine images are stored in Glance, and Ceilometer collects metrics about your overcloud.

The following diagram illustrates a physical view of how the undercloud may be hosted on one physical server and the overcloud distributed across many physical servers.



SpinalStacks Inspiration

Some key aspects of SpinalStack workflow have been incorporated into TripleO, providing options to perform introspection, benchmarking and role matching of your hardware prior to deploying OpenStack.

Hardware introspection features enable you to collect data about the properties of your hardware prior to deployment, such that specific classes of hardware may be matched to specific roles (e.g. a special hardware configuration for Compute or Storage roles). There is also the option to enable performance benchmarking during this phase, such that outliers which do not match the expected performance profile may be excluded from the deployment.

TripleO also configures servers in a similar way to SpinalStack, using stable community puppet implementations, applied in a series of steps, such that granular control and validation of the deployment is possible

2.1.2 Benefits

Using TripleOs combination of OpenStack components, and their APIs, as the infrastructure to deploy and operate OpenStack itself delivers several benefits:

- TripleOs APIs are the OpenStack APIs. Theyre well maintained, well documented, and come with client libraries and command line tools. Users who invest time in learning about TripleOs APIs are also learning about OpenStack itself, and users who are already familiar with OpenStack will find a great deal in TripleO that they already understand.
- Using the OpenStack components allows more rapid feature development of TripleO than might otherwise be the case; TripleO automatically inherits all the new features which are added to Glance, Heat etc., even when the developer of the new feature didnt explicitly have TripleO in mind.
- The same applies to bug fixes and security updates. When OpenStack developers fix bugs in the common components, those fixes are inherited by TripleO.
- Users can invest time in integrating their own scripts and utilities with TripleOs APIs with some confidence. Those APIs are cooperatively maintained and developed by the OpenStack community. Theyre not at risk of being suddenly changed or retired by a single controlling vendor.
- For developers, tight integration with the OpenStack APIs provides a solid architecture, which has gone through extensive community review.

It should be noted that not everything in TripleO is a reused OpenStack element.

2.1.3 Deployment Workflow Overview

1. Environment Preparation
    - Prepare your environment (baremetal or virtual)
    - Install undercloud
2. Undercloud Data Preparation
    - Create images to establish the overcloud
    - Register hardware nodes with undercloud
    - Introspect hardware

    - Create flavors (node profiles)
3. Deployment Planning
    - Configure overcloud roles
        - Assign flavor (node profile to match desired hardware specs)
        - Assign image (provisioning image)
        - Size the role (how many instances to deploy)
    - Configure service parameters
    - Create a Heat template describing the overcloud (auto-generated from above)
4. Deployment
    - Use Heat to deploy your template
    - Heat will use Nova to identify and reserve the appropriate nodes
    - Nova will use Ironic to startup nodes and install the correct images
5. Per-node Setup
    - When each node of the overcloud starts it will gather its configuration metadata from Heat Template configuration files
    - Hiera files are distributed across all nodes and Heat applies puppet manifests to configure the services on the nodes
    - Puppet runs in multiple steps, so that after each step there can be tests triggered to check progress of the deployment and allow easier debugging.
6. Overcloud Initialization
    - Services on nodes of the overcloud are registered with Keystone

2.1.4 Deployment Workflow Detail

Environment Preparation

In the first place, you need to check that your environment is ready. TripleO can deploy OpenStack into baremetal as well as virtual environments. You need to make sure that your environment satisfies minimum requirements for given environment type and that networking is correctly set up.

Next step is to install the undercloud. We install undercloud using Instacks script and it calls puppet scripts in the background.

For development or proof of concept (PoC) environments, Quickstart can also be used.

Undercloud Data Preparation

Images

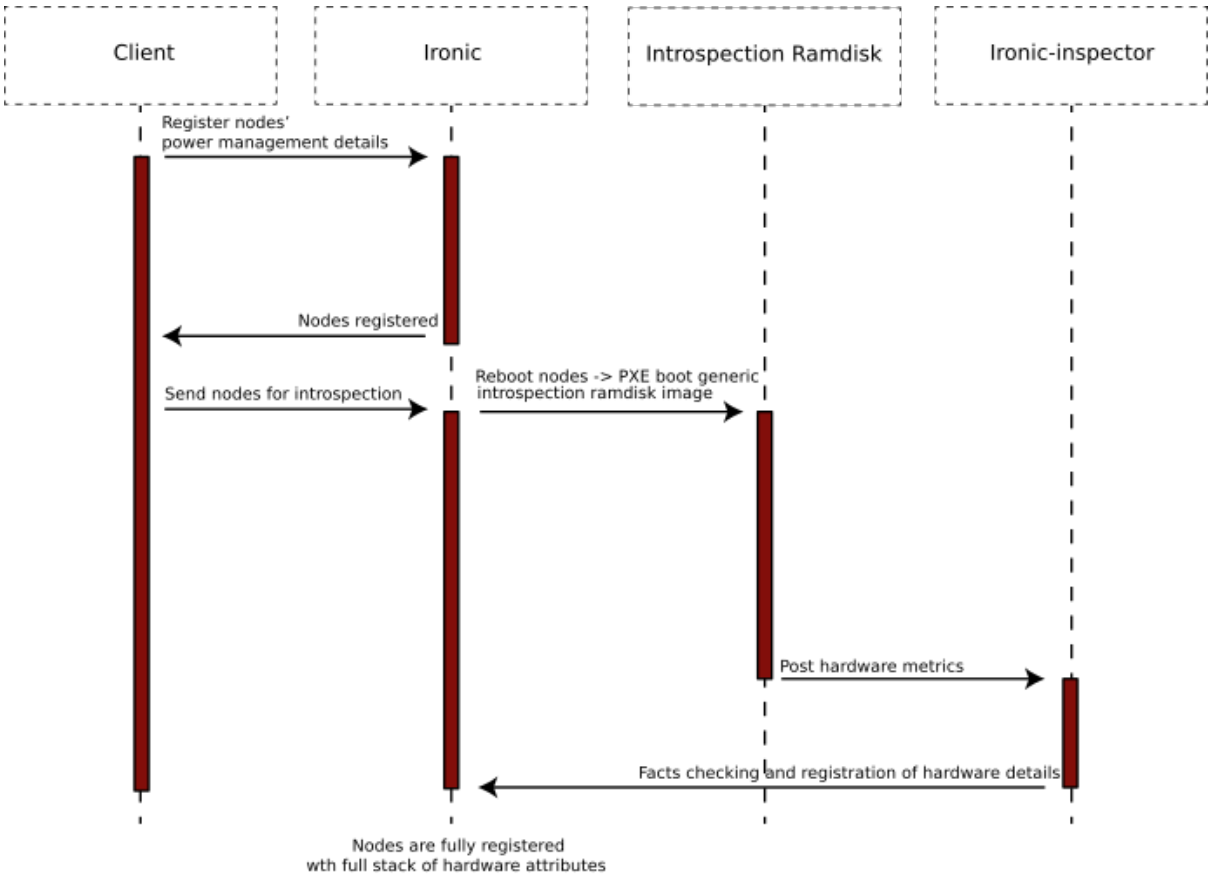
Before deploying the overcloud, you must first download or build images which will be installed on each of the nodes of the overcloud. TripleO uses diskimage-builder for building these so called Golden Images. The diskimage-builder tool takes a base image e.g. CentOS 7 and then layers additional software via configuration scripts (called elements) on top of that. The final result is a qcow2 formatted image with software installed but not configured.

While the diskimage-builder repository provides operating-system specific elements, ones specific to OpenStack, e.g. nova-api, are found in tripleo-image-elements. You can add different elements to an image to provide specific applications and services. Once all the images required to deploy the overcloud are built, they are stored in Glance running on the undercloud.

Nodes

Deploying the overcloud requires suitable hardware. The first task is to register the available hardware with Ironic, OpenStacks equivalent of a hypervisor for managing baremetal servers. Users can define the hardware attributes (such as number of CPUs, RAM, disk) manually or he can leave the fields out and run introspection of the nodes afterwards.

The sequence of events is pictured below:



- The user, via the command-line tools, or through direct API calls, registers the power management credentials for a node with Ironic.

- The user then instructs Ironic to reboot the node.
- Because the node is new, and not already fully registered, there are no specific PXE-boot instructions for it. In that case, the default action is to boot into an introspection ramdisk
- The introspection ramdisk probes the hardware on the node and gathers facts, including the number of CPU cores, the local disk size and the amount of RAM.
- The ramdisk posts the facts to the ironic-inspector API.
- All facts are passed and stored in the Ironic database.
- There can be performed advanced role matching via the ahc-match tool, which simply adds an additional role categorization to Ironic based on introspected node facts and specified conditions.

Flavors

When users are creating virtual machines (VMs) in an OpenStack cloud, the flavor that they choose specifies the capacity of the VM which should be created. The flavor defines the CPU count, the amount of RAM, the amount of disk space etc. As long as the cloud has enough capacity to grant the users wish, and the user hasnt reached their quota limit, the flavor acts as a set of instructions on exactly what kind of VM to create on the users behalf.

In the undercloud, where the machines are usually physical rather than virtual (or, at least, pre-existing, rather than created on demand), flavors have a slightly different effect. Essentially, they act as a constraint. Of all of the introspected hardware, only nodes which match a specified flavor are suitable for a particular role. This can be used to ensure that the large machines with a great deal of RAM and CPU capacity are used to run Nova in the overcloud, and the smaller machines run less demanding services, such as Keystone.

TripleO is capable of handling flavors in two different modes.

The simpler PoC (Proof of Concept) mode is intended to enable new users to experiment, without worrying about matching hardware profiles. In this mode, theres one single, global flavor, and any hardware can match it. That effectively removes flavor matching. Users can use whatever hardware they wish.

For the second mode, named Scale because it is suited to larger scale overcloud deployments, flavor matching is in full effect. A node will only be considered suitable for a given role if the role is associated with a flavor which matches the capacity of the node. Nodes without a matching flavor are effectively unusable.

This second mode allows users to ensure that their different hardware types end up running their intended role, though requires either manual node tagging or using introspection rules to tag nodes (see Profile Matching).

Deployment Planning

Whole part of planning your deployment is based on concept of **overcloud roles**. A role brings together following things:

- An image; the software to be installed on a node
- A flavor; the size of node suited to the role
- A size; number of instances which should be deployed having given role
- A set of heat templates; instructions on how to configure the node for its task

In the case of the Compute role:

- the image must contain all the required software to boot an OS and then run the KVM hypervisor and the Nova compute service
- the flavor (at least for a deployment which isnt a simple proof of concept), should specify that the machine has enough CPU capacity and RAM to host several VMs concurrently
- the Heat templates will take care of ensuring that the Nova service is correctly configured on each node when it first boots.

Currently, the roles in TripleO are very prescriptive, and in particular individual services cannot easily be scaled independently of the Controller role (other than storage nodes). More flexibility in this regard is planned in a future release.

Customizable things during deployment planning are:

- Number of nodes for each role
- Service parameters configuration
- Network configuration (NIC configuration options, isolated vs. single overlay)
- Ceph rbd backend options and defaults
- Ways to pass in extra configuration, e.g site-specific customizations

Deployment

Deployment to physical servers happens through a collaboration of Heat, Nova, Neutron, Glance and Ironic.

The Heat templates and environments are served to Heat which will orchestrate the whole deployment and it will create a stack. Stack is Heats own term for the applications that it creates. The overcloud, in Heat terms, is a particularly complex instance of a stack.

In order for the stack to be deployed, Heat makes successive calls to Nova, OpenStacks compute service controller. Nova depends upon Ironic, which, as described above has acquired an inventory of introspected hardware by this stage in the process.

At this point, Nova flavors may act as a constraint, influencing the range of machines which may be picked for deployment by the Nova scheduler. For each request to deploy a new node with a specific role, Nova filters the list of available nodes, ensuring that the selected nodes meet the hardware requirements.

Once the target node has been selected, Ironic does the actual provisioning of the node, Ironic retrieves the OS image associated with the role from Glance, causes the node to boot a deployment ramdisk and then, in the typical case, exports the nodes local disk over iSCSI so that the disk can be partitioned and the have the OS image written onto it by the Ironic Conductor.

See Ironics Understanding Baremetal Deployment for further details.

Per-node Setup

TBD - Puppet

2.1.5 High Availability (HA)

TripleO will use Pacemaker to achieve high-availability.

Reference architecture document: https://github.com/beekhof/osp-ha-deploy

Note: Current HA solution is being developed by our community.

2.1.6 Managing the Deployment

After the overcloud deployment is completed, it will be possible to monitor, scale it out or perform basic maintenance operations via the CLI.

Monitoring the Overcloud

When the overcloud is deployed, Ceilometer can be configured to track a set of OS metrics for each node (system load, CPU utilization, swap usage etc.)

Additionally, Ironic exports IPMI metrics for nodes, which can also be stored in Ceilometer. This enables checks on hardware state such as fan operation/failure and internal chassis temperatures.

The metrics which Ceilometer gathers can be queried for Ceilometers REST API, or by using the command line client.

Note: There are plans to add more operational tooling to the future release.

Scaling-out the Overcloud

The process of scaling out the overcloud by adding new nodes involves these stages:

- Making sure you have enough nodes to deploy on (or register new nodes as described in the Undercloud Data Preparation section above).
- Calling Heat to update the stack which will apply the set of changes to the overcloud.

CHAPTER

THREE

TRIPLEO COMPONENTS

3.1 TripleO Components

This section contains a list of components that TripleO uses. The components are organized in categories, and include a basic description, useful links, and contribution information.

3.1.1 Shared Libraries

diskimage-builder

diskimage-builder is an image building tool. It is used by `openstack overcloud image build`.

How to contribute

See the diskimage-builder README.rst for a further explanation of the tooling. Submit your changes via OpenStack Gerrit (see OpenStack Developers Guide).

Useful links

- Upstream Project Documentation: https://docs.openstack.org/diskimage-builder/
- Bugs: https://bugs.launchpad.net/diskimage-builder
- Git repository: https://opendev.org/openstack/diskimage-builder/

dib-utils

dib-utils contains tools that are used by diskimage-builder.

How to contribute

Submit your changes via OpenStack Gerrit (see OpenStack Developers Guide).

Useful links

- Bugs: https://bugs.launchpad.net/diskimage-builder
- Git repository: https://opendev.org/openstack/dib-utils/

os-*-config

The os-*-config projects are a suite of tools used to configure instances deployed via TripleO. They include:

- os-collect-config
- os-refresh-config
- os-apply-config
- os-net-config

How to contribute

Each tool uses tox to manage the development environment. Submit your changes via OpenStack Gerrit (see OpenStack Developers Guide).

Useful links

- Bugs:
  - os-collect-config: https://bugs.launchpad.net/os-collect-config
  - os-refresh-config: https://bugs.launchpad.net/os-refresh-config
  - os-apply-config: https://bugs.launchpad.net/os-apply-config
  - os-net-config: https://bugs.launchpad.net/os-net-config
- Git repositories:
  - os-collect-config: https://opendev.org/openstack/os-collect-config
  - os-refresh-config https://opendev.org/openstack/os-refresh-config
  - os-apply-config https://opendev.org/openstack/os-apply-config
  - os-net-config https://opendev.org/openstack/os-net-config

tripleo-image-elements

tripleo-image-elements is a repository of diskimage-builder style elements used for installing various software components.

How to contribute

Submit your changes via OpenStack Gerrit (see OpenStack Developers Guide).

Useful links

- Git repository: https://opendev.org/openstack/tripleo-image-elements

3.1.2 Installer

instack

instack executes diskimage-builder style elements on the current system. This enables a current running system to have an element applied in the same way that diskimage-builder applies the element to an image build.

instack, in its current form, should be considered low level tooling. It is meant to be used by higher level scripting that understands what elements and hook scripts need execution. Using instack requires a rather in depth knowledge of the elements within diskimage-builder and tripleo-image-elements.

How to contribute

Submit your changes via OpenStack Gerrit (see OpenStack Developers Guide).

Useful links

- Git repository: https://opendev.org/openstack/instack
- Bugs: https://launchpad.net/tripleo

instack-undercloud

instack-undercloud is a TripleO style undercloud installer based around instack.

How to contribute

Submit your changes via OpenStack Gerrit (see OpenStack Developers Guide).

Useful links

- Git repository: https://opendev.org/openstack/instack-undercloud
- Bugs: https://launchpad.net/tripleo

3.1.3 Node Management

ironic

Ironic project is responsible for provisioning and managing bare metal instances.

For testing purposes Ironic can also be used for provisioning and managing virtual machines which act as bare metal nodes via special driver `pxe_ssh`.

How to contribute

Ironic uses tox to manage the development environment, see the Developer Quick-Start, Ironic Developer Guidelines and OpenStack Developers Guide for details.

Useful links

- Upstream Project: https://docs.openstack.org/ironic/index.html
- Bugs: https://bugs.launchpad.net/ironic
- Blueprints: https://blueprints.launchpad.net/ironic
  - Specs process should be followed for suggesting new features.

- Approved Specs: http://specs.openstack.org/openstack/ironic-specs/

ironic inspector (former ironic-discoverd)

Ironic Inspector project is responsible for inspection of hardware properties for newly enrolled nodes (see also *ironic*).

How to contribute

Ironic Inspector uses tox to manage the development environment, see upstream documentation for details.

Useful links

- Upstream Project: https://github.com/openstack/ironic-inspector
- PyPI: https://pypi.org/project/ironic-inspector
- Bugs: https://bugs.launchpad.net/ironic-inspector

VirtualBMC

A helper command to translate IPMI calls into libvirt calls. Used for testing bare metal provisioning on virtual environments.

How to contribute

VirtualBMC uses tox to manage the development environment in a similar way to Ironic.

Useful links

- Source: https://opendev.org/openstack/virtualbmc
- Bugs: https://bugs.launchpad.net/virtualbmc

3.1.4 Deployment & Orchestration

heat

Heat is OpenStacks orchestration tool. It reads YAML files describing the OpenStack deployments resources (machines, their configurations etc.) and gets those resources into the desired state, often by talking to other components (e.g. Nova).

How to contribute

- Use devstack with Heat to set up a development environment. Submit your changes via OpenStack Gerrit (see OpenStack Developers Guide).

Useful links

- Upstream Project: https://wiki.openstack.org/wiki/Heat
- Bugs: https://bugs.launchpad.net/heat
- Blueprints: https://blueprints.launchpad.net/heat

heat-templates

The heat-templates repository contains additional image elements for producing disk images ready to be configured by Puppet via Heat.

How to contribute

- Use devtest with Puppet to set up a development environment. Submit your changes via OpenStack Gerrit (see OpenStack Developers Guide).

Useful links

- Upstream Project: https://opendev.org/openstack/heat-templates
- Bugs: https://bugs.launchpad.net/heat-templates
- Blueprints: https://blueprints.launchpad.net/heat-templates

tripleo-heat-templates

The tripleo-heat-templates describe the OpenStack deployment in Heat Orchestration Template YAML files and Puppet manifests. The templates are deployed via Heat.

How to contribute

- Use devtest with Puppet to set up a development environment. Submit your changes via OpenStack Gerrit (see OpenStack Developers Guide).

Useful links

- Upstream Project: https://opendev.org/openstack/tripleo-heat-templates
- Bugs: https://bugs.launchpad.net/tripleo
- Blueprints: https://blueprints.launchpad.net/tripleo

nova

nova provides a cloud computing fabric controller.

How to contribute

- Read the Development Quickstart to set up a development environment. Submit your changes via OpenStack Gerrit (see OpenStack Developers Guide).

Useful links

- Git repository: https://opendev.org/openstack/nova
- Bugs: https://bugs.launchpad.net/nova
- Blueprints: https://blueprints.launchpad.net/nova

puppet-*

The OpenStack Puppet modules are used to configure the OpenStack deployment (write configuration, start services etc.). They are used via the tripleo-heat-templates.

How to contribute

- Use devtest with Puppet to set up a development environment. Submit your changes via OpenStack Gerrit (see OpenStack Developers Guide).

Useful links

- Upstream Project: https://wiki.openstack.org/wiki/Puppet

tripleo-puppet-elements

The tripleo-puppet-elements describe the contents of disk images which TripleO uses to deploy OpenStack. Its the same kind of elements as in tripleo-image-elements, but tripleo-puppet-elements are specific for Puppet-enabled images.

How to contribute

- Use devtest with Puppet to set up a development environment. Submit your changes via OpenStack Gerrit (see OpenStack Developers Guide).

Useful links

- Upstream Project: https://opendev.org/openstack/tripleo-puppet-elements
- Bugs: https://bugs.launchpad.net/tripleo
- Blueprints: https://blueprints.launchpad.net/tripleo

3.1.5 User Interfaces

python-openstackclient

The python-openstackclient is an upstream CLI tool which can manage multiple openstack services. It wraps openstack clients like glance, nova, etc. and maps them under intuitive names like openstack image, compute, etc.

The main value is that all services can be controlled by a single (openstack) command with consistent syntax and behaviour.

How to contribute

- python-openstackclient uses tox to manage the development environment, see the python-openstackclient documentation for details. Submit your changes via OpenStack Gerrit (see OpenStack Developers Guide).

Useful links

- Upstream Project: https://opendev.org/openstack/python-openstackclient
- Bugs: https://bugs.launchpad.net/python-openstackclient
- Blueprints: https://blueprints.launchpad.net/python-openstackclient

- Human interface guide: https://docs.openstack.org/python-openstackclient/humaninterfaceguide.html

python-tripleoclient

The python-tripleoclient is a CLI tool embedded into python-openstackclient. It provides functions related to instack installation and initial configuration like node introspection, overcloud image building and uploading, etc.

How to contribute

- python-tripleoclient uses tox to manage the development environment, see the python-tripleoclient documentation for details. Submit your changes via Gerrit.

Useful links

- Project: https://opendev.org/openstack/python-tripleoclient

tripleo-ui

TripleO UI is the web interface for TripleO.

How to contribute

- See the documentation for details.

Useful links

- Bugs: https://bugs.launchpad.net/tripleo-ui
- Blueprints: https://blueprints.launchpad.net/tripleo-ui

3.1.6 tripleo-validations

Pre and post-deployment validations for the deployment workflow.

Useful links

- Upstream Project: https://opendev.org/openstack/tripleo-validations/
- Bugs: https://bugs.launchpad.net/tripleo/+bugs?field.tag=validations
- Documentation for individual validations: https://docs.openstack.org/tripleo-validations/latest/readme.html#existing-validations

---

Note: When reporting an issue, make sure you add the `validations` tag.

3.1.7 Deprecated

Tuskar

The Tuskar project was responsible for planning the deployments and generating the corresponding Heat templates. This is no longer necessary as Heat supports this composability out of the box.

The source code is available below, but please note that it should not be used for new deployments.

https://github.com/openstack/tuskar

CHAPTER

FOUR

TRIPLEO CI GUIDE

4.1 TripleO CI Guide

4.1.1 TripleO CI jobs primer

This primer aims to demonstrate where the Triple ci jobs are defined and illustrate the difference between the check and gate queues and how jobs are executed in them. Which queue a job is executed in also affects whether the job is defined as voting or not. Generally:

- new jobs are run in check and are non voting
- once a job is voting in check, it needs to be added to gate too.
- once a job is voting in check and gate you should add it to the promotion jobs so that tripleo promotions (i.e. from tripleo-testing to current-tripleo) will depend on successful execution of that job.

Once a job becomes voting it must be added to the gate queue too. If it isnt then we may end up with a situation where something passes the voting check job and merges without being run in the gate queue. It could be that for some reason it would have failed in the gate and thus not have merged. A common occurrence is the check jobs run on a particular submission and pass on one day but then not actually merge (and so run in the gate) until much later perhaps even after some days.In the meantime some unrelated change merges in another project which would cause the job to fail in the gate, but since were not running it there the code submission merges. This then means that the job is broken in subsequent check runs.

Non tripleo-projects are not gated in tripleo. The promotion jobs represent the point at which we take the latest built tripleo packages and the latest built non-tripleo projects packages (like nova, neutron etc) and test these together. For more information about promotions refer to *Promotion Stages*

Where do tripleo-ci jobs live

Note: If you ever need to search for a particular job to see which file it is defined in or which tripleo project repos it is running for you can search by name in the openstack-codesearch (e.g. that is a search for the tripleo-ci-centos-7-scenario003-standalone job).

Note: If you ever want to see the status for a particular job with respect to how often it is failing or passing, you can check the zuul_builds status and search by job name (again the linked example is for

scenario003-standalone).

The tripleo ci jobs live in the tripleo-ci repo and specifically in various files defined under the zuul.d directory. As an example we can examine one of the scenario-standalone-jobs:

```
- job:
  name: tripleo-ci-centos-7-scenario001-standalone
  voting: true
  parent: tripleo-ci-base-standalone
  nodeset: single-centos-7-node
  branches: ^(?!stable/(newton|ocata|pike|queens|rocky)).*$
  vars:
    featureset: '052'
    standalone_ceph: true
    featureset_override:
      standalone_container_cli: docker
      standalone_environment_files:
        - 'ci/environments/scenario001-standalone.yaml'
        - 'environments/low-memory-usage.yaml'
      tempest_plugins:
        - python-telemetry-tests-tempest
        - python-heat-tests-tempest
      test_white_regex: ''
      tempest_workers: 1
      tempest_extra_config: {'telemetry.alarm_granularity': '60'}
      tempest_whitelist:
        - 'tempest.api.identity.v3'
        - 'tempest.scenario.test_volume_boot_pattern.TestVolumeBootPattern.
↪test_volume_boot_pattern'
        - 'telemetry_tempest_plugin.scenario.test_telemetry_integration.
↪TestTelemetryIntegration'
```

As you can see the job definition consists of the unique job name followed by the rest of the zuul variables, including whether the job is voting and which node layout (nodeset) should be used for that job. The unique job name is then used in the zuul layout (discussed in the next section) to determine if the job is run in check or gate or both. Since the job shown above is set as voting we can expect it to be defined in both gate and check.

Zuul queues - gate vs check

As with all OpenStack projects there are two zuul queues to which jobs are scheduled - the check jobs which are run each time a change is submitted and then the gate jobs which are run before a change is merged. There is also an experimental queue but that is invoked manually.

Which queue a given job is run in is determined by the zuul layout file for the given project - e.g. here is tripleo-heat-templates-zuul-layout. The layout file has the following general format:

```
- project:
   templates:
   .. list of templates
```

(continues on next page)

(continued from previous page)

```
  check:
    jobs:
      .. list of job names and any options for each
  gate:
    queue: tripleo
    jobs:
      .. list of job names and any options for each
```

The templates: section in the outline above is significant because the layout can also be defined in one of the included templates. For example the scenario-standalone-layout defines the check/gate layout for the tripleo-standalone-scenarios-full template which is then included by the projects that want the jobs defined in that template to execute in the manner it specifies.

Where do tripleo promotion jobs live

Note: If you even need to find the definition for a particular promotion job you can search for it by name using the rdo-codesearch.

The tripleo promotions jobs are not defined in the tripleo-ci but instead live in the rdo-jobs repository. For more information about the promotion pipeline in TripleO refer to the *Promotion Stages*

Similar to the tripleo-ci jobs, they are defined in various files under the rdo-jobs-zuul.d directory and the job definitions look very similar to the tripleo-ci ones - for example the periodic-tripleo-ci-centos-7-multinode-1ctlr-featureset010-master:

```
- job:
  name: periodic-tripleo-ci-centos-7-multinode-1ctlr-featureset010-master
  parent: tripleo-ci-base-multinode-periodic
  vars:
    nodes: 1ctlr
    featureset: '010'
    release: master
```

If you even need to find the definition for a particular promotion job you can search for it by name using the rdo-codesearch.

Contacting CI team

When in need you can contact the TripleO CI team members on one of the two irc channels on OFTC #tripleo by mentioning `@oooq` keyword in your message as team members get notified about such messages. It is good to remember that those nicknames with `|ruck` and `|rover` suffix are on duty to look for CI status.

4.1.2 Reproduce CI jobs for debugging and development

Knowing that at times (perhaps always) manipulating zuul jobs to do your bidding can be frustrating. Perhaps you are trying to reproduce a bug, test a patch, or just bored on a Sunday afternoon. I wanted to briefly remind folks of their options.

RDOs zuul:

RDOs zuul is setup to directly inherit from upstream zuul. Any TripleO job that executes upstream should be re-runnable in RDOs zuul. A distinct advantage here is that you can ask RDO admins to hold the job for you, get your ssh keys on the box and debug the live environment. Its good stuff. To hold a node, ask your friends in #rhos-ops

Use testproject: Some documentation can be found here:

upstream job example:

```
- project:
    name: testproject
    check:
      jobs:
        - tripleo-ci-centos-8-content-provider
        - tripleo-ci-centos-8-containers-multinode:
            dependencies:
              - tripleo-ci-centos-8-content-provider

    gate:
      jobs: []
```

periodic job, perhaps recreating a CIX issue example:

```
- project:
   name: testproject
   check:
      jobs:
         - tripleo-ci-centos-8-scenario002-standalone:
            vars:
               timeout: 22000
         - periodic-tripleo-ci-centos-8-standalone-full-tempest-scenario-
↪master:
            vars:
               timeout: 22000
               force_periodic: true
         - periodic-tripleo-ci-centos-8-standalone-full-tempest-scenario-
↪victoria:
            vars:
               timeout: 22000
```

(continues on next page)

(continued from previous page)

```
            force_periodic: true
        - periodic-tripleo-ci-centos-8-standalone-full-tempest-scenario-
↪ussuri:
            vars:
              timeout: 22000
              force_periodic: true

  gate:
    jobs: []
```

Remember that depends-on can bring in any upstream changes.

- Here is an example commit message:

Test jobs with new ovn package

```
Test jobs with new ovn package

Depends-On: https://review.opendev.org/c/openstack/openstack-tempest-skiplist/
↪+/775493

Change-Id: I7b392acc4690199caa78cac90956e717105f4c6e
```

Local zuul:

Setting up zuul and friends locally is a much heavier lift than your first option. Instructions and scripts to help you are available in any upstream TripleO job, and here

A basic readme for the logs can be found directly in the logs directory of any tripleo job.

- Basic Readme
- Job reproduce

If you are familiar w/ zuul and friends, containers, etc.. this could be a good option for you and your team. There are a lot of moving parts and its complicated, well because its complicated. A good way to become more familiar with zuul would be to try out zuuls tutorial

zuul-runner:

A long hard fought battle of persuasion and influence has been fought with the maintainers of the zuul project. The blueprints and specs have merged. The projects status is not complete as there are many unmerged patches to date.

Other Options:

Finally, if you are not attempting to recreate, test, play with an upstream tripleo job and just want to develop code there is another option. A lot of developers find tripleo-lab to be quite useful. Many devels have their own patterns as well, what works for you is fine.

Summary:

For what its worth imho using testproject jobs is an efficient, low barrier to getting things done with upstream TripleO jobs. Ill be updating the documentation and references to try and help over the next few days, patches are welcome :)

4.1.3 How to add a TripleO job to your projects check pipeline

To ensure a non-TripleO projects changes work with TripleO an additional check job can be added to the projects job definitions in OpenStacks project config

Project Config Example

In this case well use openstack/neutron as an example to understand how this works. Note that this is only an example and this job may not be appropriate for your project, we will cover how to pick a job later on in this documentation. Browse through the layout.yaml file in the project-config repository until you find:

```
- name: openstack/neutron
  template:
    - name: merge-check
    - ...
    - ...
  check:
    - ...
    - ...
    - gate-tripleo-ci-centos-7-nonha-multinode-oooq-nv
```

The above configuration will run the TripleO job `gate-tripleo-ci-centos-7-nonha-multinode-oooq-nv` without voting (nv). This type of job is used to inform the reviewers of the patch whether or not the change under review works with TripleO.

How to pick which job to execute for any given OpenStack project

TripleO can deploy a number of different OpenStack services. To best utilize the available upstream CI resources TripleO uses the same concept as the puppet-openstack-integration project to define how services are deployed. The TripleO documentation regarding services can be found here. Review the TripleO documentation and find a scenario that includes the services that your project requires to be tested. Once you have determined which scenario to use you are ready to pick a TripleO check job.

The following is a list of available check jobs:

```
gate-tripleo-ci-centos-7-scenario001-multinode-oooq
gate-tripleo-ci-centos-7-scenario001-multinode-oooq-puppet
gate-tripleo-ci-centos-7-scenario001-multinode-oooq-container
gate-tripleo-ci-centos-7-scenario002-multinode-oooq
gate-tripleo-ci-centos-7-scenario002-multinode-oooq-puppet
gate-tripleo-ci-centos-7-scenario002-multinode-oooq-container
gate-tripleo-ci-centos-7-scenario003-multinode-oooq
gate-tripleo-ci-centos-7-scenario003-multinode-oooq-puppet
gate-tripleo-ci-centos-7-scenario003-multinode-oooq-container
gate-tripleo-ci-centos-7-scenario004-multinode-oooq
gate-tripleo-ci-centos-7-scenario004-multinode-oooq-puppet
gate-tripleo-ci-centos-7-scenario004-multinode-oooq-container
gate-tripleo-ci-centos-7-nonha-multinode-oooq
gate-tripleo-ci-centos-7-containers-multinode
```

Note over time additional scenarios will be added and will follow the same pattern as the job names listed above.

Adding a new non-voting check job

Find your project in layout.yaml. An example of a project will look like the following example:

```
- name: openstack/$project
  template:
    - ...
    - ...
```

Note `$project` is the name of your project.

Under the section named `check`, add the job that best suits your project. Be sure to add `-nv` to the job name to ensure the job does not vote:

```
check:
  - ...
  - ...
  - $job-nv
```

Enabling voting jobs

If your project is interested in gating your project with a voting version of a TripleO job, you can follow the openstack/mistral projects example in layout.yaml

For example:

```
- name: openstack/mistral
  template:
    -name: merge-check
    - ...
    - ...
  check:
```

(continues on next page)

(continued from previous page)

```
    - ...
    - ...
    - gate-tripleo-ci-centos-7-scenario003-multinode-oooq-puppet
  gate:
    - gate-tripleo-ci-centos-7-scenario003-multinode-oooq-puppet
```

Note the example does **not** append `-nv` as a suffix to the job name

Troubleshooting a failed job

When your newly added job fails, you may want to download its logs for a local inspection and root cause analysis. Use the tripleo-ci getthelogs script for that.

Enabling tempest tests notification

There is a way to get notifications by email when a job finishes to running tempest. People interested to receive these notifications can submit a patch to add their email address in this config file. Instructions can be found here.

featureset override

In TripleO CI, we test each patchset using different jobs. These jobs are defined using featureset config files. Each featureset config file is mapped to a job template that is defined in tripleo-ci. Tempest tests are basically triggered in scenario jobs in order to post validate the a particular scenario deployment. The set of tempest tests that run for a given TripleO CI job is defined in the featureset config files. You may want to run a popular TripleO CI job with a custom set of Tempest tests and override the default Tempest run. This can be accomplished through adding the *featureset_overrides* var to zuul job config *vars:* section. The allowed featureset_override are defined in the tripleo-ci run-test role. This setting allows projects to override featureset post deployment configuration. Some of the overridable settings are:

- *run_tempest*: To run tempest or not (true|false).
- *tempest_whitelist*: List of tests you want to be executed.
- *test_black_regex*: Set of tempest tests to skip.
- *tempest_format*: To run tempest using different format (packages, containers, venv).
- *tempest_extra_config*: A dict of additional tempest config to be overridden.
- *tempest_plugins*: A list of tempest plugins needs to be installed.
- *standalone_environment_files*: List of environment files to be overridden by the featureset configuration on standalone deployment. The environment file should exist in tripleo-heat-templates repo.
- *test_white_regex*: Regex to be used by tempest
- *tempest_workers*: Numbers of parallel workers to run
- *standalone_container_cli*: Container cli to use
- *tempest_private_net_provider_type*: The Neutron type driver that should be used by tempest tests.

For a given job *tripleo-ci-centos-7-scenario001-multinode-oooq-container*, you can create a new abstract layer job and overrides the tempest tests:

```
- job:
    name: scn001-multinode-oooq-container-custom-tempest
    parent: tripleo-ci-centos-7-scenario001-multinode-oooq-container
    ...
    vars:
      featureset_override:
        run_tempest: true
        tempest_whitelist:
          - 'tempest.scenario.test_volume_boot_pattern.TestVolumeBootPattern.
↪test_volume_boot_pattern'
        test_black_regex:
          - 'keystone_tempest_plugin'
        tempest_format: 'containers'
        tempest_extra_config: {'compute-feature-enabled.attach_encrypted_
↪volume': 'True',
                               'auth.tempest_roles': '"Member"'}
        tempest_plugins:
          - 'python2-keystone-tests-tempest'
          - 'python2-cinder-tests-tempest'
        tempest_workers: 1
        test_white_regex:
          - 'tempest.api.identity'
          - 'keystone_tempest_plugin'
        standalone_environment_files:
          - 'environments/low-memory-usage.yaml'
          - 'ci/environments/scenario003-standalone.yaml'
        standalone_container_cli: docker
```

In a similar way, for skipping Tempest run for the scenario001 job, you can do something like:

```
- job:
    name: scn001-multinode-oooq-container-skip-tempest
    parent: tripleo-ci-centos-7-scenario001-multinode-oooq-container
    ...
    vars:
      featureset_override:
        run_tempest: false
```

Below is the list of jobs based on *tripleo-puppet-ci-centos-7-standalone* which uses featureset_override and run specific tempest tests against puppet projects:

- puppet-nova
  - job name: puppet-nova-tripleo-standalone
  - tempest_test: compute
- puppet-horizon
  - job name: puppet-horizon-tripleo-standalone
  - tempest_test: horizon

- puppet-keystone
  - job name: puppet-keystone-tripleo-standalone
  - tempest_test: keystone_tempest_plugin & identity
- puppet-glance
  - job name: puppet-glance-tripleo-standalone
  - tempest_test: image
- puppet-cinder
  - job name: puppet-cinder-tripleo-standalone
  - tempest_test: volume & cinder_tempest_tests
- puppet-neutron
  - job name: puppet-neutron-tripleo-standalone
  - tempest_test: neutron_tempest_tests & network
- puppet-swift
  - job name: puppet-swift-tripleo-standalone
  - tempest_test: object_storage

4.1.4 Standalone Scenario jobs

This section gives an overview and some details on the standalone scenario ci jobs. The standalone deployment is intended as a one node development environment for TripleO. - see the Standalone Deploy Guide for more information on setting up a standalone environment.

A scenario is a concept used in TripleO to describe a collection of services - see the service-testing-matrix for more information about each scenario and the services deployed there. We combine the two to define the standalone scenario jobs.

These are intended to give developers faster feedback (the jobs are relatively fast to complete) and allow us to have better coverage across services by defining a number of scenarios. Crucially the standalone scenario jobs allow us to increase coverage without further increasing our resource usage footprint with eachjob only taking a single node. See this openstack-dev-thread for background around the move from the multinode jobs to the more resource friendly standalone versions.

Where

The standalone scenario jobs (hereafter referred to as just standalone in this document), are defined in the tripleo-ci/zuul.d/standalone.yaml file. Besides the definitions for each of the scenario00X-standalone jobs, this file also carries the tripleo-standalone-scenarios-full_project-template which defines the zuul layout and files: sections for the standalone jobs in a central location.

Thus, the jobs are consumed by other projects across tripleo by inclusion of the template in their respective zuul layout file, for example tripleo-heat-templates and tripleo-common.

Besides the job definitions in the tripleo-ci repo, the other main part of the standalone jobs is a service environment file, which lives in the tripleo-heat-templates-ci/environments. As you can see in scenario001-env, scenario002-env, scenario003-env and scenario004-env that is where we define the services and parameters that are part of a given scenario.

How

The standalone jobs are special in that they differ from traditional multinode jobs by having a shared featureset rather than requiring a dedicated featureset for each job. Some of the standalone scenarios, notably scenario012 will end up having a dedicated-featureset however in most cases the base standalone-featureset052 can be re-used for the different scenarios. Notably you can see that scenario001-job, scenario002-job, scenario003-job and scenario004-job job definitions are all using the same standalone-featureset052.

Given that we use the same featureset the main differentiator between these standalone jobs is the scenario environment file, which we pass using featureset_override (see *How to add a TripleO job to your projects check pipeline*). For example in the scenario001 job we point to the scenario001-standalone.yaml (scenario001-env):

```
- job:
    name: tripleo-ci-centos-7-scenario001-standalone
    voting: true
    parent: tripleo-ci-base-standalone
    nodeset: single-centos-7-node
    branches: ^(?!stable/(newton|ocata|pike|queens|rocky)).*$
    vars:
      featureset: '052'
      standalone_ceph: true
      featureset_override:
        standalone_container_cli: docker
        standalone_environment_files:
          - 'environments/low-memory-usage.yaml'
          - 'ci/environments/scenario001-standalone.yaml'
   ...
```

Finally we use a task in the tripleo-ci-run-test-role to pass the scenario environment file into the standalone deployment command using the standalone role standalone_custom_env_files parameter.

4.1.5 Baremetal jobs

This section gives an overview and some details on the baremetal CI jobs. The baremetal deployment is intended as a multinode real world production-like environment for TripleO. - see Baremetal deploy guide for more information on setting up a baremetal environment.

The baremetal jobs, previously running in the RDO Phase 2 of the promotion pipeline from Jenkins servers, now are triggered from an internal Software Factory instance of Zuul. These promotion jobs testing containers built on tripleo-ci-testing hashes run on real baremetal hardware, report to dlrn and can be included in the TripleO promotion criteria.

The goal is to give developers feedback on real deployments and allow us to have better coverage on issues seen in production environments. It also allows an approximation of OVB jobs running in RDO cloud in order to get an apples-to-apples comparison to eliminate infra issues.

Where

The hardware is maintained internally and cannot be accessed by upstream Zuul or RDO Cloud. The internal Software Factory instance provides a version of infra upstream tools as Zuul, Gerrit and Nodepool for running the defined baremetal jobs. Refer to Software Factory Documentation for more details.

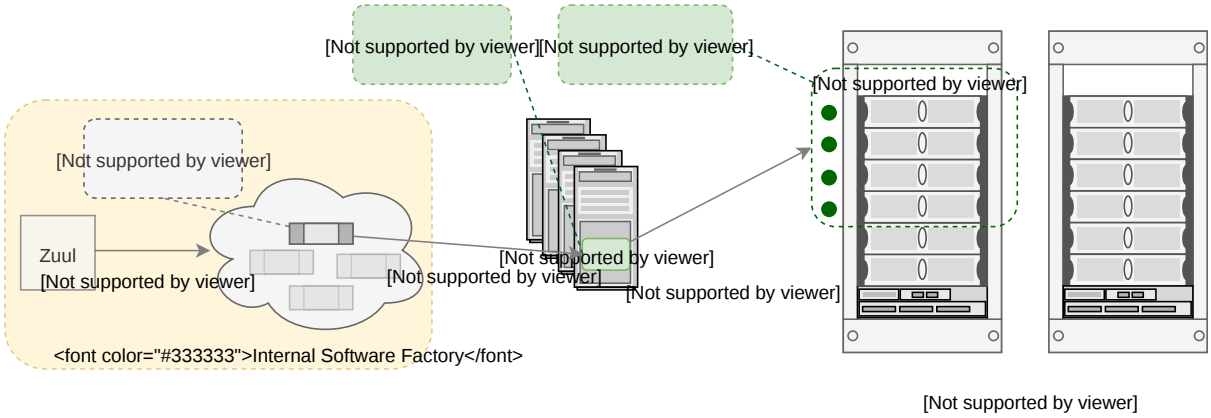
The jobs will use hardware_environments/<env name>/instackenv.json file and the hardware_environments/<env name>/network_configs/single_nic_vlans settings file. These configurations are explored in more detail below.

How

The baremetal job workflow is described as follows:

1. The baremetal jobs are triggered in the periodic pipeline and initially run on a Nodepool node that can be called as executor, where the job starts executing its playbooks and roles.
2. The job sshs to the baremetal machine which will host the undercloud vm and creates a new vm on which the undercloud will be installed and set up.
3. Finally the undercloud VM deploys the overcloud on real baremetal nodes defined in the instackenv.json configuration over pxe boot.

This workflow for baremetal jobs is illustrated in the following figure:



Parenting from upstream and RDO repos

Jobs that run from internal Zuul can parent off, and use resources (jobs, roles etc.) from, upstream (review.opendev.org) and RDO (review.rdoproject.org) repos. As such, duplication can be kept to a minimum and jobs that run internally on baremetal hardware can maintain parity with OVB jobs run in RDO Cloud.

For example, a base TripleO CI job in Zuul

```
- job:
    name: tripleo-ci-base-baremetal
    abstract: true
    description: |
       Base abstract job for Baremetal TripleO
    parent: tripleo-ci-base
```

(continues on next page)

(continued from previous page)

```
  nodeset: tripleo-baremetal-centos-7-primary
  attempts: 1
  required-projects:
    - rdo-jobs
  roles:
    - zuul: rdo-jobs
  pre-run:
    - playbooks/configure-mirrors.yaml
    - playbooks/copy-env-vars-baremetal.yaml
  vars:
    # must be overridden
    undercloud: <undercloud>
    environment_infra: baremetal
    environment_type: baremetal
    playbooks:
    - baremetal-prep-virthost.yml
    - baremetal-full-undercloud.yml
    - baremetal-full-overcloud-prep.yml
    - baremetal-full-overcloud.yml
    - baremetal-full-overcloud-validate.yml
    tags:
    - all
```

Now adding the dlrn reporting

```
- job:
  name: tripleo-ci-base-baremetal-dlrn
  parent: tripleo-ci-base-baremetal
  abstract: true
  description: |
    Base abstract job to do DLRN reporting
  required-projects:
    - config
  roles:
    - zuul: config
  pre-run:
    - playbooks/dlrn/pre-dlrn.yaml
  post-run:
    - playbooks/dlrn/post-dlrn.yaml
  secrets:
    - dlrnapi
```

Example of a specific hardware job in Zuul:

Note that multiple jobs cannot be run on the hardware concurrently. The base job is modified to include semaphore https://zuul-ci.org/docs/zuul/user/config.html#semaphore to run each only one at a time

```
- job:
  name: tripleo-ci-base-baremetal-dlrn-my_env
  abstract: true
```

(continues on next page)

(continued from previous page)

```
  parent: tripleo-ci-base-baremetal-dlrn
  vars:
    baremetal_env_vars: >-
      {{ local_working_dir }}/hardware_environments/my_env/<truncated_path>/
↪env_settings.yml
    undercloud: <my_env-undercloud-baremetal-host-address>
  semaphore:
    name: my_env

- job:
  name: periodic-tripleo-ci-centos-7-baremetal-3ctlr_1comp-featureset001-
↪master
  parent: tripleo-ci-base-baremetal-dlrn-my_env
  vars:
    nodes: 3ctlr_1comp
    featureset: '001'
    release: master
```

Hardware Settings

An example of hardware settings for baremetal environment my_env is shown below:

hardware_environments / my_env / network_configs / single_nic_vlans / env_settings.yml

```
environment_type: my_env

# undercloud.conf settings
undercloud_network_cidr: 10.10.10.0/26
undercloud_local_ip: 10.10.10.1/26
undercloud_network_gateway: 10.10.10.100
undercloud_undercloud_public_vip: 10.10.10.2
undercloud_undercloud_admin_vip: 10.10.10.3
undercloud_local_interface: eth1
undercloud_masquerade_network: 10.10.10.0/26
undercloud_dhcp_start: 10.10.10.5
undercloud_dhcp_end: 10.10.10.24
undercloud_inspection_iprange: 10.10.10.25,10.10.10.39
undercloud_undercloud_nameservers: 10.10.10.200
network_isolation_ipv4_cidr: 10.10.10.64/26
undercloud_external_network_cidr: 10.10.10.64/26

# undercloud vm settings
virthost_provisioning_interface: eno2
virthost_provisioning_ip: 10.10.10.4
virthost_provisioning_netmask: 255.255.255.192
virthost_provisioning_hwaddr: FF:FF:FF:FF:FF:FF
virthost_ext_provision_interface: eno1

undercloud_memory: 28672
```

(continues on next page)

(continued from previous page)

```
undercloud_disk: 80
undercloud_vcpu: 8

undercloud_instackenv_template: >-
  {{ local_working_dir }}/hardware_environments/my_env/instackenv.json

undercloud_type: virtual
step_introspect: true
introspect: true

# network-environment.yaml settings
network_environment_args:
InternalApiNetCidr: 172.21.33.0/24
StorageNetCidr: 172.21.36.0/24
StorageMgmtNetCidr: 172.21.35.0/24
TenantNetCidr: 172.16.0.0/24
ExternalNetCidr: 10.10.10.64/26
BondInterfaceOvsOptions: "mode=4 lacp_rate=fast"
InternalApiAllocationPools: [{'start': '172.21.33.10', 'end': '172.21.33.
↪200'}]
StorageAllocationPools: [{'start': '172.21.36.10', 'end': '172.21.36.200'}
↪]
StorageMgmtAllocationPools: [{'start': '172.21.35.10', 'end': '172.21.35.
↪200'}]
TenantAllocationPools: [{'start': '172.16.0.10', 'end': '172.16.0.200'}]
# Leave room for floating IPs starting at .128
ExternalAllocationPools: [{'start': '10.10.10.101', 'end': '10.10.10.120'}
↪]
ExternalInterfaceDefaultRoute: 10.10.10.130
InternalApiNetworkVlanID: 1272
StorageNetworkVlanID: 1273
StorageMgmtNetworkVlanID: 1274
ExternalNetworkVlanID: 113
TenantNetworkVlanID: 1275
NeutronExternalNetworkBridge: "''"
PublicVirtualFixedIPs: [{"ip_address": "10.10.10.90"}]
ControlPlaneSubnetCidr: "26"
ControlPlaneDefaultRoute: 10.10.10.1
EC2MetadataIp: 10.10.10.1
DnsServers: ["8.8.8.8", "8.8.4.4"]
NtpServer: ["216.239.35.12","time.google.com","0.north-america.pool.ntp.
↪org"]

step_root_device_size: false
step_install_upstream_ipxe: false
hw_env: my_env
enable_vbmc: false
```

hardware_environments / my_env / instackenv.json

```
{
"nodes": [
   {
      "pm_password": "<passwd>",
      "pm_type": "ipmi",
      "mac": [
      "FF:FF:FF:FF:FF:FF"
      ],
      "cpu": "12",
      "memory": "32768",
      "disk": "558",
      "arch": "x86_64",
      "pm_user": "Administrator",
      "pm_addr": "10.1.1.11"
   },
   {
      "pm_password": "<passwd>",
      "pm_type": "ipmi",
      "mac": [
      "FF:FF:FF:FF:FF:FF"
      ],
      "cpu": "12",
      "memory": "32768",
      "disk": "558",
      "arch": "x86_64",
      "pm_user": "Administrator",
      "pm_addr": "10.1.1.12"
   },
   {
      "pm_password": "<passwd>",
      "pm_type": "ipmi",
      "mac": [
      "FF:FF:FF:FF:FF:FF"
      ],
      "cpu": "12",
      "memory": "32768",
      "disk": "558",
      "arch": "x86_64",
      "pm_user": "Administrator",
      "pm_addr": "10.1.1.13"
   },
   {
      "pm_password": "<passwd>",
      "pm_type": "ipmi",
      "mac": [
      "FF:FF:FF:FF:FF:FF"
      ],
      "cpu": "12",
      "memory": "32768",
      "disk": "558",
```

(continues on next page)

(continued from previous page)

```
        "arch": "x86_64",
        "pm_user": "Administrator",
        "pm_addr": "10.1.1.14"
    }
]
}
```

4.1.6 How the TripleO-RDO Pipelines Promotions Work

Building consumable RDO repos and images involves various stages. Each stage takes inputs and outputs artifacts. This document explains the stages comprising the promotion pipelines, and the tools used to create and manage the resulting artifacts.

What is DLRN?

DLRN is a tool to build RPM packages from each commit to a set of OpenStack-related git repositories that are included in RDO. DLRN builds are run through CI and to detect packaging issues with the upstream branches of these Openstack projects.

DLRN Artifacts - Hashes and Repos

When a DLRN build completes, it produces a new hash and related repo version. For example, the Pike builds on CentOS are available at: https://trunk.rdoproject.org/centos7-pike/. The builds are placed in directories by DLRN hash. Each directory contains the RPMs as well as a repo file https://trunk.rdoproject.org/centos7-pike/current-tripleo/delorean.repo and a `commit.yaml` file https://trunk.rdoproject.org/centos7-pike/current-tripleo/commit.yaml.

There are some standard links that are updated as the builds complete and pass stages of CI. Examples are these links are:

- https://trunk.rdoproject.org/centos7-pike/current/
- https://trunk.rdoproject.org/centos7-pike/consistent/
- https://trunk.rdoproject.org/centos7-pike/current-tripleo/
- https://trunk.rdoproject.org/centos7-pike/current-tripleo-rdo/
- https://trunk.rdoproject.org/centos7-pike/current-tripleo-rdo-internal/
- https://trunk.rdoproject.org/centos7-pike/tripleo-ci-testing/

The above links will be referenced in the sections below.

Promoting through the Stages - DLRN API

DLRN API Client

The DLRN API client enables users to query repo status, upload new hashes and create promotions. Calls to the dlrnapi_client retrieve the inputs to stages and upload artifacts after stages.

For example:

```
$ dlrnapi --url https://trunk.rdoproject.org/api-centos-master-uc \
  promotion-get --promote-name tripleo-ci-testing

[{'commit_hash': 'ec650aa2c8ce952e4a33651190301494178ac562',
 'distro_hash': '9a7acc684265872ff288a11610614c3b5739939b',
 'promote_name': 'tripleo-ci-testing',
 'timestamp': 1506427440},
 {'commit_hash': 'ec650aa2c8ce952e4a33651190301494178ac562',
[..]


$ dlrnapi --url https://trunk.rdoproject.org/api-centos-master-uc \
  repo-status --commit-hash ec650aa2c8ce952e4a33651190301494178ac562 \
 --distro-hash 9a7acc684265872ff288a11610614c3b5739939b

[{'commit_hash': 'ec650aa2c8ce952e4a33651190301494178ac562',
 'distro_hash': '9a7acc684265872ff288a11610614c3b5739939b',
 'in_progress': False,
 'job_id': 'consistent',
 'notes': '',
 'success': True,
 'timestamp': 1506409403,
 'url': ''},
 {'commit_hash': 'ec650aa2c8ce952e4a33651190301494178ac562',
 'distro_hash': '9a7acc684265872ff288a11610614c3b5739939b',
 'in_progress': False,
 'job_id': 'periodic-singlenode-featureset023',
 'notes': '',
 'success': True,
 'timestamp': 1506414726,
 'url': 'https://logs.rdoproject.org/openstack-periodic-4hr/periodic-tripleo-
↪centos-7-master-containers-build/8a76883'},
 {'commit_hash': 'ec650aa2c8ce952e4a33651190301494178ac562',
[..]
```

DLRN API Promoter

The DLRN API Promoter script is a Python script that, based on the information in an input config file, will promote an existing DLRN link to another link, provided the required tests return successful results.

For example, the master ini config file is passed to the promoter script to promote the `current-tripleo` link to `current-tripleo-rdo`. See the sections above where both these links (for Pike) were shown.

In the RDO Phase 1 pipeline, the tests listed under the `[current-tripleo-rdo]` are run with the `current-tripleo` hash. Each test reports its `success` status to the DLRN API endpoint for the Master release, `api-centos-master-uc`.

If each test reports `SUCCESS: true`, the content of the `current-tripleo` will become the new content of the `current-tripleo-rdo` hash.

For complete documentation on how to run the Promoter script see: https://github.com/rdo-infra/ci-config/blob/master/ci-scripts/dlrnapi_promoter/README.md

Pushing RDO containers to `docker.io`

The DLRN Promoter script calls the container push playbook to push the RDO containers at each stage to docker.io. Note that the above `docker.io` link shows containers tagged with `tripleo-ci-testing`, `current-tripleo` and `current-tripleo-rdo`.

DLRN API Promoter Server

It is recommended that the Promoter script is run from a dedicated server. The promoter-setup repo contains the Ansible playbook used to setup the promoter-server in the RDO Cloud environment. This playbook allows the promoter script server to be rebuilt as required.

4.1.7 TripleO CI Promotions

This section introduces the concept of promotions in TripleO. In short, a promotion happens when we can certify the latest version of all packages required for a TripleO deployment of OpenStack as being in a good state and without regressions.

The certification consists of running Zuul CI jobs with the latest packages built from source for TripleO code (list of TripleO repos at[1]) and the latest packages built from source for non-tripleo code. If the tests are successful, then the result is certified as **current-tripleo**, ready to be consumed by the TripleO CI check and gate jobs (see[2] for more information about check and gate).

This process is continuous as new code is merged into the various repos. Every time we get a successful completion of the promotion CI jobs, the tested content is promoted to be the new **current-tripleo**, hence the name this workflow is known by. At a given time, the latest **current-tripleo** is the baseline by which we test all new code submissions to the TripleO project.

[1] List of TripleO repos
[2] TripleO Check and Gate jobs

TripleO vs non-tripleo repos

All proposed code submissions across the various tripleo repos are gated by the TripleO community which owns and manages the zuul check and gate jobs for those repos.

However, we cannot gate changes to anything outside TripleO, including all the OpenStack projects used by TripleO as well as any dependencies such as Open vSwitch or Pacemaker.

Even though we cannot gate on those external repos, the promotion process allows us to test our TripleO code with their latest versions. If there are regressions or any other bugs (and assuming ideal test coverage) the promotion jobs will fail accordingly allowing the TripleO CI team to investigate and file launchpad bugs so the issue(s) can be addressed.

RDO DLRN & Promotion Criteria

TripleO CI jobs consume packages built by the RDO DLRN service (delorean) so we first introduce it here. An overview is given on the RDO project site at[3].

In short, RDO DLRN builds RPMs from source and publishes the resulting packages and repos. Each build or repo is identifiable using a unique build ID.

RDO DLRN assigns named tags to particular build IDs. You can see all of these named tags by browsing at the RDO DLRN package root, for example for Centos8 master branch at[4]. Of particular importance to the TripleO promotion workflow are:

```
* current
* consistent
* component-ci-testing
* promoted-components
* tripleo-ci-testing
* current-tripleo
```

The list of tags in the order given above gives the logical progression through the TripleO promotion workflow.

The build ID referenced by each of those named tags is constantly updated as new content is promoted to become the new named tag.

A general pattern in DLRN is that **current** is applied to the very latest build, that is, the latest commits to a particular repo. A new **current** build is generated periodically (e.g. every half hour). The **consistent** tag represents the latest version of packages where there were no errors encountered during the build for any of those (i.e. all packages were built successfully). The **consistent** build is what TripleO consumes as the entry point to the TripleO promotion workflow.

One last point to be made about RDO DLRN is that after the TripleO promotion CI jobs are executed against a particular DLRN build ID, the results are reported back to DLRN. For example, you can query using the build ID at[5] to get the list of jobs that were executed against that specific content, together with the results for each.

The list of jobs that are required to pass before we can promote a particular build is known as the promotion criteria. In order to promote, TripleO queries the DLRN API to get the results for a particular

[3] RDO DLRN Overview @ rdoproject.org
[4] Index of RDO DLRN builds for Centos 8 master @ rdoproject.org
[5] Query RDO DLRN by build ID @ rdoproject.org

build and compares the passing jobs to the promotion criteria, before promoting or rejecting that content accordingly. You can find the master centos8 promotion criteria at[6] for example.

The TripleO Promotion Pipelines

A pipeline refers to a series of Zuul CI jobs and what we refer to as the TripleO promotion workflow is actually a number of interconnected pipelines. At the highest level conceptually these are grouped into either *Component* or *Integration* pipelines. The output of the Component pipeline serves as input to the Integration pipeline.

A Component is a conceptual grouping of packages related by functional area (with respect to an OpenStack deployment). This grouping is enforced in practice by the RDO DLRN server and the current list of all components can be found at[7]. For example, you can expect to find the openstack-nova- packages within the Compute component.

The Component pipeline actually consists of a number of individual pipelines, one for each of the components. The starting point for each of these is the latest **consistent** build of the component packages and we will go into more detail about the flow inside the component pipelines in the following section.

A successful run of the jobs for the given component allows us to certify that content as being the new **promoted-components**, ready to be used as input to the Integration pipeline. The Integration pipeline qualifies the result of the components tested together and when that is successful we can promote to a new current-tripleo. This is shown conceptually for a subset of components here:

In the diagram above, you can see the component pipeline at the top with the compute, cinder and security components. This feeds into the integration pipeline in the bottom half of the diagram where promoted-components will be tested together and if successful produce the new **current-tripleo**.

The Component Promotion Pipeline

As noted above, the Component pipeline is actually a series of individual pipelines, one for each component. While these all operate and promote in the same way, they do so independently of each other. So the latest **compute/promoted-components** may be much newer than the latest **security/promoted-components**, if the latter is failing to promote for example. The following flowchart shows the progression of the RDO DLRN tags through a single component pipeline while in practice this flow is repeated in parallel per component.

As illustrated above, the entry point to the component pipelines is the latest **consistent** build from RDO DLRN. Once a day a periodic job tags the latest **consistent** build as **component-ci-testing**. For example you can see the history for the baremetal component job at[8] descriptively named **periodic-tripleo-centos-8-master-component-baremetal-promote-consistent-to-component-ci-testing**.

After this job has completed the content marked as **component-ci-testing** becomes the new candidate for promotion to be passed through the component CI jobs. The **component-ci-testing** repo content is tested with the latest **current-tripleo** repos of everything else. Remember that at a given time **current-tripleo** is the known good baseline by which we test all new content and the same applies to new content tested in the component pipelines.

As an example of the component CI jobs, you can see the history for the baremetal component standalone job at[9]. If you navigate to the *logs/undercloud/etc/yum.repos.d/* directory for one of those job runs you

[6] Centos8 current-tripleo promotion criteria at time of writing
[7] Centos8 RDO DLRN components @ rdoproject.org
[8] Zuul job history periodic-tripleo-centos-8-master-component-baremetal-promote-consistent-to-component-ci-testing
[9] Zuul job history periodic-tripleo-ci-centos-8-standalone-baremetal-master

will see (at least) the following repos:

- delorean.repo - which provides the latest current-tripleo content
- baremetal-component.repo - which provides the component-ci-testing content that we are trying to promote.

You may notice that the trick allowing the baremetal-component.repo to have precedence for the packages it provides is to set the repo priority accordingly (*1* for the component and *20* for delorean.repo).

Another periodic job checks the result of the **component-ci-testing** job runs and if the component promotion criteria is satisfied the candidate content is promoted and tagged as the new **promoted-components**. You can find the promotion criteria for Centos8 master components at[10].

As an example the history for the zuul job that handles promotion to promoted-components for the cinder component can be found at[11]

You can explore the latest content tagged as **promoted-components** for the compute component at[12]. All the component **promoted-components** are aggregated into one repo that can be found at[13] and looks like the following:

```
[delorean-component-baremetal]
name=delorean-openstack-ironic-9999119f737cd39206df3d73e23e5f47933a6f32
baseurl=https://trunk.rdoproject.org/centos8/component/baremetal/99/99/
↪9999119f737cd39206df3d73e23e5f47933a6f32_1b0aff0d
enabled=1
gpgcheck=0
priority=1

[delorean-component-cinder]
name=delorean-openstack-cinder-482e6a3cc5cca697b54ee1d853a4eca6e6f3cfc7
baseurl=https://trunk.rdoproject.org/centos8/component/cinder/48/2e/
↪482e6a3cc5cca697b54ee1d853a4eca6e6f3cfc7_ae00ff8c
enabled=1
gpgcheck=0
priority=1
```

Every time a component promotes a new **component/promoted-components** the aggregated **promoted-components** delorean.repo on the RDO DLRN server is updated with the new content.

This **promoted-components** repo is used as the starting point for the TripleO Integration promotion pipeline.

[10] Centos8 master promoted-components promotion critiera at time of writing
[11] Zuul job history periodic-tripleo-centos-8-master-component-cinder-promote-to-promoted-components
[12] Compute promoted-components @ rdoproject.org
[13] Centos8 master promoted-components delorean.repo @ rdoproject.org

The Integration Promotion Pipeline

The Integration pipeline as the name suggests is the integration point where we test new content from all components together. The consolidated **promoted-components** delorean.repo produced by the component pipeline is tested with a series of CI jobs. If the jobs listed in the promotion criteria pass, we promote that content and tag it as **current-tripleo**.

As can be seen in the flowchart above, the **promoted-components** content is periodically promoted (pinned) to **tripleo-ci-testing**, which becomes the new promotion candidate to be tested. You can find the build history for the job that promotes to **tripleo-ci-testing** for Centos 8 master, descriptively named **periodic-tripleo-centos-8-master-promote-promoted-components-to-tripleo-ci-testing**, at[14].

First the **tripleo-ci-testing** content is used to build containers and overcloud deployment images and these are pushed to RDO cloud to be used by the rest of the jobs in the integration pipeline.

The periodic promotion jobs are then executed with the results being reported back to DLRN. If the right jobs pass according to the promotion criteria then the **tripleo-ci-testing** content is promoted and tagged to become the new **current-tripleo**.

An important distinction in the integration pipeline compared to the promotion pipeline is in the final promotion of content. In the component pipeline the **promoted-components** content is tagged by a periodic Zuul job as described above. For the Integration pipeline however, the promotion to **current-tripleo** happens with the use of a dedicated service. This service is known to the tripleo-ci squad by a few names including the promotion server, the promoter server and the promoter.

In short the promoter periodically queries delorean for the results of the last few tripleo-ci-testing runs. It compares the results to the promotion criteria and if successful it re-tags the container and overcloud deployment images as **current-tripleo** and pushes back to RDO cloud (as well as to the quay.io and docker registries). It also talks to the DLRN server and retags the successful **tripleo-ci-testing** repo as the new **current-tripleo**. You can read more about the promoter with links to the code at[15].

References

4.1.8 emit-releases-file and releases.sh

The emit-releases-file tool is a python script that lives in the tripleo-ci repo under the scripts/emit_releases_file directory. This script produces an output file called *releases.sh* containing shell variable export commands. These shell variables set the release **name** and **hash** for the installation and target (versions) of a given job. For example, installing latest stable branch (currently stein) and upgrading to master. The **hash** is the delorean repo hash from which the packages used in the job are to be installed.

The contents of *releases.sh* will differ depending on the type of upgrade or update operation being performed by a given job and this is ultimately determined by the featureset. Each upgrade or update related featureset sets boolean variables that signal the type of upgrade performed. For example featureset050 is used for undercloud upgrade and it sets:

```
undercloud_upgrade: true
```

The *releases.sh* for an undercloud upgrade job looks like:

[14] Zuul job history periodic-tripleo-centos-8-master-promote-promoted-components-to-tripleo-ci-testing
[15] TripleO CI docs Promotion Server and Criteria

```
#!/bin/env bash
export UNDERCLOUD_INSTALL_RELEASE="stein"
export UNDERCLOUD_INSTALL_HASH="c5b283cab4999921135b3815cd4e051b43999bce_
↪5b53d5ba"
export UNDERCLOUD_TARGET_RELEASE="master"
export UNDERCLOUD_TARGET_HASH="be90d93c3c5f77f428d12a9a8a2ef97b9dada8f3_
↪5b53d5ba"
export OVERCLOUD_DEPLOY_RELEASE="master"
export OVERCLOUD_DEPLOY_HASH="be90d93c3c5f77f428d12a9a8a2ef97b9dada8f3_
↪5b53d5ba"
export OVERCLOUD_TARGET_RELEASE="master"
export OVERCLOUD_TARGET_HASH="be90d93c3c5f77f428d12a9a8a2ef97b9dada8f3_
↪5b53d5ba"
export STANDALONE_DEPLOY_RELEASE="master"
export STANDALONE_DEPLOY_HASH="be90d93c3c5f77f428d12a9a8a2ef97b9dada8f3_
↪5b53d5ba"
export STANDALONE_DEPLOY_NEWEST_HASH=
↪"b4c2270cc6bec2aaa3018e55173017c6428237a5_3eee5076"
export STANDALONE_TARGET_RELEASE="master"
export STANDALONE_TARGET_NEWEST_HASH=
↪"b4c2270cc6bec2aaa3018e55173017c6428237a5_3eee5076"
export STANDALONE_TARGET_HASH="be90d93c3c5f77f428d12a9a8a2ef97b9dada8f3_
↪5b53d5ba"
```

As can be seen there are three different groups of keys set: *UNDERCLOUD_INSTALL* and *UNDERCLOUD_TARGET* is one group, then *OVERCLOUD_DEPLOY* and *OVERCLOUD_TARGET*, and finally *STANDALONE_DEPLOY* and *STANDALONE_TARGET*. For each of those groups we have the *_RELEASE* name and delorean *_HASH*. Since the example above is generated from an undercloud upgrade job/featureset only the undercloud related values are set correctly. The values for *OVERCLOUD_* and *STANDALONE_* are set to the default values with both *_DEPLOY* and *_TARGET* referring to *master*.

Where is releases.sh used

The releases script is not used for all CI jobs or even for all upgrades related jobs. There is a conditional in the tripleo-ci run-test role which determines the list of jobs for which we *use emit-releases-file*. In future we may remove this conditional altogether.

Once it is determined that the releases.sh file will be used, a list of extra RELEASE_ARGS is compiled to be passed into the subsequent quickstart playbook invocations. An example of what these *RELEASE_ARGS* looks like is:

```
--extra-vars @/home/zuul/workspace/.quickstart/config/release/tripleo-ci/
↪CentOS-7/master.yml -e dlrn_hash=be90d93c3c5f77f428d12a9a8a2ef97b9dada8f3_
↪5b53d5ba -e get_build_command=be90d93c3c5f77f428d12a9a8a2ef97b9dada8f3_
↪5b53d5ba
```

The *RELEASE_ARGS* are resolved by a helper function get_extra_vars_from_release. As you can see this function uses the release name passed in via the *_RELEASE* value from the *releases.sh* to set the right release configuration file from the tripleo-quickstart config/release/ directory which sets variables for the ansible execution. It also sets the *dlrn_hash* which is used to setup the right repo and thus versions

of packages and finally the get_build_command is used to make sure we have the right containers for the job.

As you can see in the list of compiled *RELEASE_ARGS* the *INSTALL* or *TARGET* are passed in to the get_extra_vars_from_release function, depending on the playbook:

```
declare -A RELEASE_ARGS=(
    ["multinode-undercloud.yml"]=$(get_extra_vars_from_release \
        $UNDERCLOUD_INSTALL_RELEASE $UNDERCLOUD_INSTALL_HASH)
    ["multinode-undercloud-upgrade.yml"]=$(get_extra_vars_from_release \
        $UNDERCLOUD_TARGET_RELEASE $UNDERCLOUD_TARGET_HASH)
```

So for the multinode-undercloud.yml use INSTALL_RELEASE but for multinode-undercloud-upgrade.yml use TARGET_RELEASE and HASH.

4.1.9 TripleO CI ruck|rover primer

The tripleo-squad divides their work across 3 week sprints. During sprint planning 2 members of the team are nominated as the ruck and rover. You can easily identify these unfortunate souls in OFTC #oooq with ruck or rover in their irc nick.

In short the ruck and rover are tasked with keeping the lights on for a given TripleO CI sprint. This means:

- ensuring gate queues are green to keep TripleO patches merging.
- ensuring promotion jobs are green to keep TripleO up to date with the rest of OpenStack and everything else that isnt TripleO! Target is bugs filed + escalated + fixed for promotion at *least* once a week.

The ruck|rover concept adopted by Tripleo CI are taken from Australian Rules Football. The ruck monitors the queue and files bugs, and the rover picks up those bugs and runs with them until theyre fixed.

This document is a primer for anyone new to the TripleO CI squad or otherwise interested in how the ruck|rover of the TripleO CI squad operate. See the CI Team Structure document for general information about how the (rest of the) TripleO CI team is organised and operates in a given sprint.

Ruck

The ruck monitors the various jobs across the various tripleo related repos both upstream tripleo-ci and rdo-infra jobs and periodics for promotions. The grafana dashboard at http://cockpit-ci.tripleo.org/ is one of the tools used by the ruck to monitor jobs (and many other things, more info on grafana below).

Any new issues are triaged by collecting logs from multiple instances of the error (i.e. same error in different jobs or different runs of the same job). The ruck monitors the failing jobs and files bugs for all known or confirmed things currently affecting TripleO CI.

Launchpad is used as the bug tracker - here is a list of recently created Tripleo launchpad bugs. When filing a new bug, the ruck will add the correct milestone, change the status to Triaged add the appropriate tag(s):

- ci: a general tag for all ci related bugs - any bug about a failing CI job should have this.
- alert: critical bugs e.g. something that affects a great number of jobs. This tag causes the bug to be advertised in irc OFTC #tripleo.

- tempest: bug is tempest related - failing tests or other tempest related error.
- ci-reproducer: related to the zuul based job reproducer
- promotion-blocker: this is used when the failing job(s) is in the promotion criteria (more on that below). Bugs with this tag are picked up by a script running periodically and converted to a CIX card which are tracked twice a week in a CI Escalation Status meeting.
- ovb: bug is related to ovb (openstack-virtual-baremetal) jobs.

For the periodic promotion jobs the ruck must ensure that the jobs defined as being in promotion criteria are passing. The criteria is simply a list of jobs which must pass for a promotion to occur (see the promotion docs for more info on the promotion stages in TripleO). This list is maintained in a file per branch in the ci-config-dlrnapi-promoter-config directory. For tripleo-ci promotions we are interested in promotions from current to current-tripleo (see promotion). Thus, looking at master.yaml at time of writing for example:

```
promotions:
current-tripleo:
  candidate_label: tripleo-ci-testing
  criteria:
    # Jobs to be added as they are defined and qualified
    - periodic-tripleo-ci-build-containers-ubi-8-push
    - periodic-tripleo-centos-8-buildimage-overcloud-full-master
    - periodic-tripleo-centos-8-buildimage-overcloud-hardened-uefi-full-master
    - periodic-tripleo-centos-8-buildimage-ironic-python-agent-master
    - periodic-tripleo-ci-centos-8-standalone-master
...
```

The above means that for a promotion to happen all the jobs defined under current-tripleo must pass. Obviously this list changes over time as jobs are created and retired. It is sometimes necessary to temporarily skip a job from that list (which is why you may see some jobs commented out with #).

Rover

The rover then takes the bugs filed by the ruck and tries to fix them. That is *not* to say that the rover is expected or indeed able to fix all encountered things! Really the expectation is that the rover has a root cause, or at least understands where the bug is coming from (e.g. which service).

In some cases bugs are fixed once a new version of some service is released (and in tripleo-ci jobs after a promotion if it is a non tripleo service/project). In this case the rover is expected to know what that fix is and do everything they can to make it available in the jobs. This will range from posting gerrit reviews to bump some service version in requirements.txt through to simply harassing the right folks ;) in the relevant TripleO Squad.

In other cases bugs may be deprioritized - for example if the job is non voting or is not in the promotion criteria then any related bugs are less likely to be getting the rovers attention. If you are interested in such jobs or bugs then you should go to #OFTC oooq channel and find the folks with ruck or rover in their nick and harass them about it!

Of course for other cases there are bona fide bugs with the TripleO CI code that the rover is expected to fix. To avoid being overwhelmed time management is hugely important for the rover especially under high load. As a general rule the rover should not spend any more than half a day (or four hours) on any

particular bug. Once this threshold is passed the rover should reach out and escalate to any component experts.

Under lighter load the rover is encouraged to help with any open bugs perhaps those ongoing issues with lower priority (e.g. non-voting jobs) and even non CI bugs in TripleO or any other relevant OpenStack component.

Tools

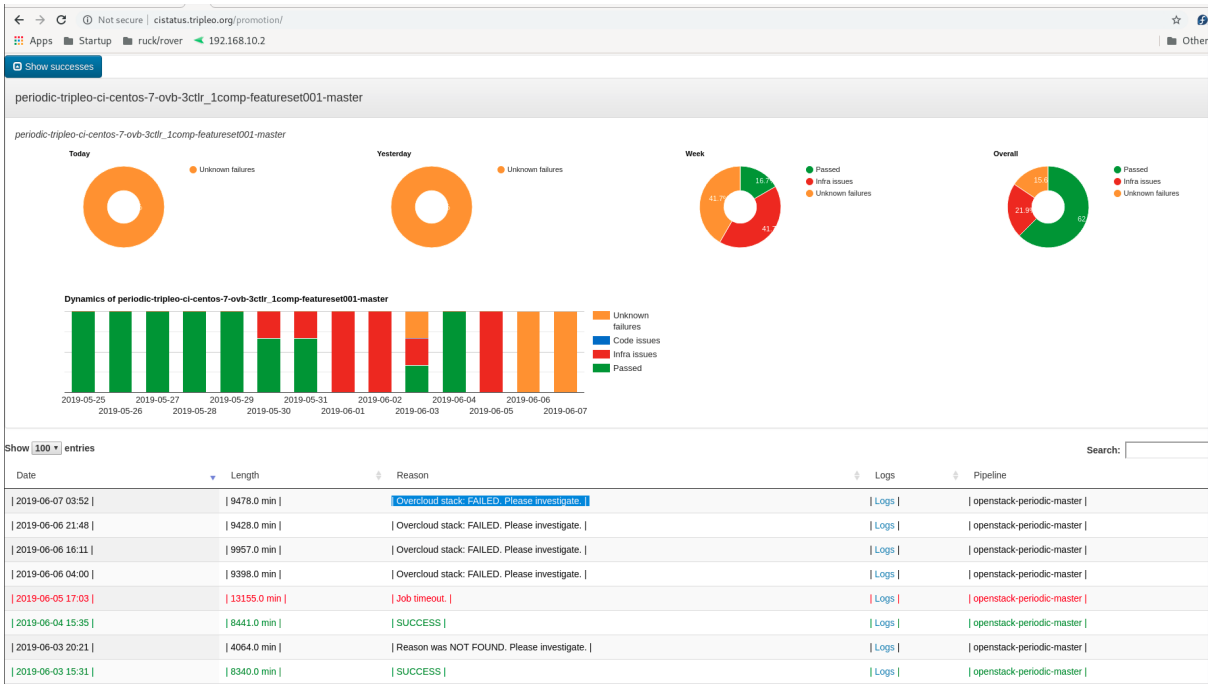
The TripleO squad has developed two main tools to help the ruck and rover do their job efficiently. They are known within the squad as grafana and sova (the names of the underlying code in each case):

- grafana: http://cockpit-ci.tripleo.org/
- sova: http://cistatus.tripleo.org/
- etherpad: $varies
- ci health: http://ci-health.tripleo.org/

The ruck|rover are encouraged to use an etherpad that is kept up to date for any ongoing issues actively being worked on. Besides allowing coordination between ruck and rover themselves (the TripleO CI team is distributed across a number of time zones) one other use case is to allow tripleo-devs to check if the reason a particular job is failing on their code review is known or if they need to go harrass the ruck|rover about it in OFTC #oooq. The location of the current ruck|rover etherpad is given in grafana (see below).

Sova

In sova you can see for each of check, gate, and promotions a list of all jobs, grouped by functionality (ovb or containers) as well as by branch in the case of promotion jobs. By clicking on a particular job you can see the most recent failures and successes with link to logs for more investigation. Sova tries to determine where and how the a job fails and reports that accordingly as shown below.



Grafana

Grafana is used to track many things and is also constantly evolving so we highlight only a few main data points here. The top of the dashboard has some meters showing the overall health of CI.

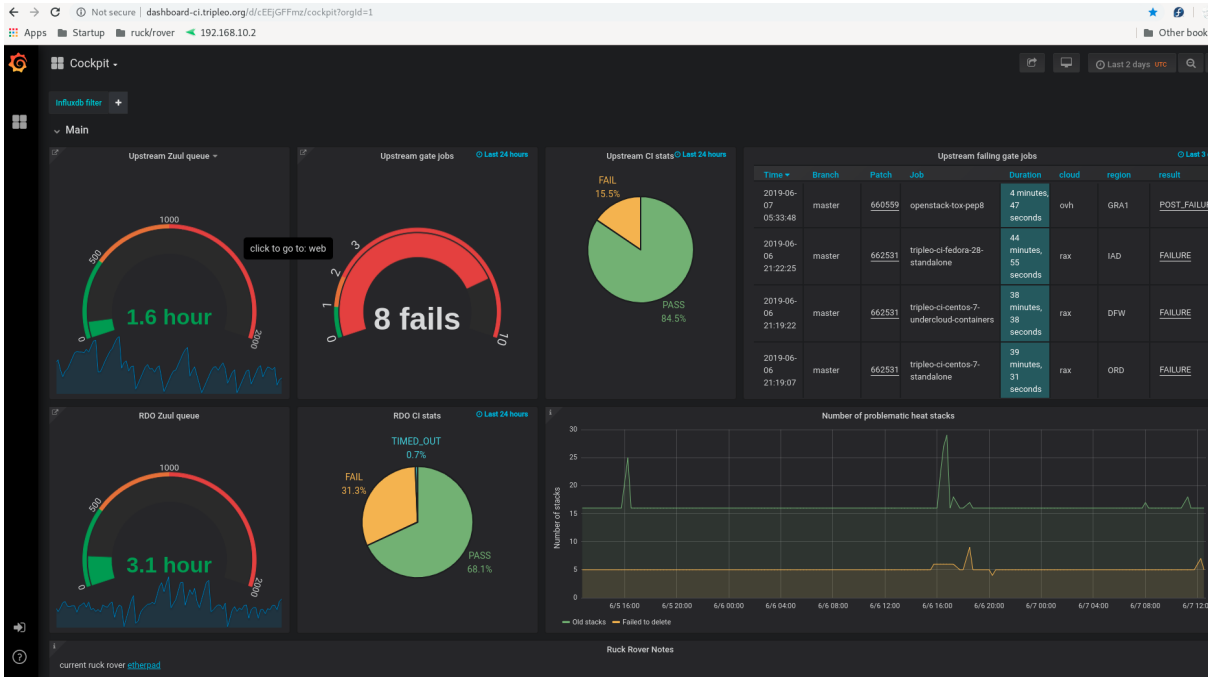
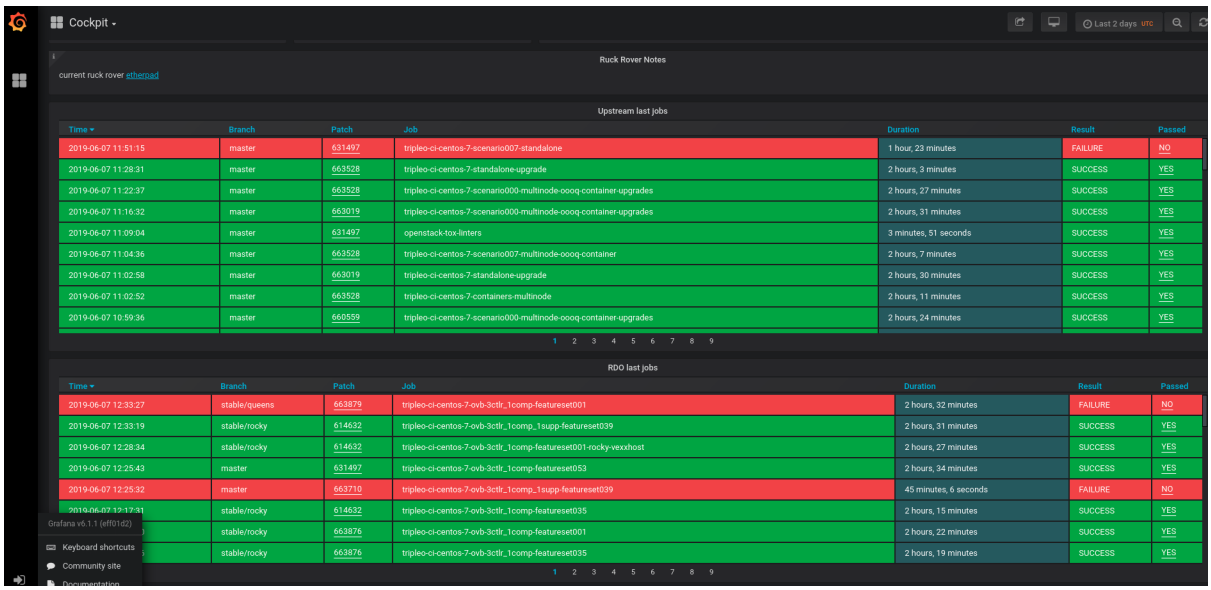


Fig. 1: As can be seen left to right - the Upstream Zuul queue gives the time a review waits before being picked up by zuul for jobs to run against it, the Upstream gate jobs shows the number of failing gate jobs in the last 24 hours, Upstream CI stats shows the ratio of passing to failing jobs as a Pie chart (anything above 80% pass is good) and finally a list of the latest failing gate jobs with links. At the bottom left there is a link to the current ruck rover etherpad.

Grafana is also useful for tracking promotions across branches.

Finally grafana tracks a list of all running jobs highlighting the failures in red.



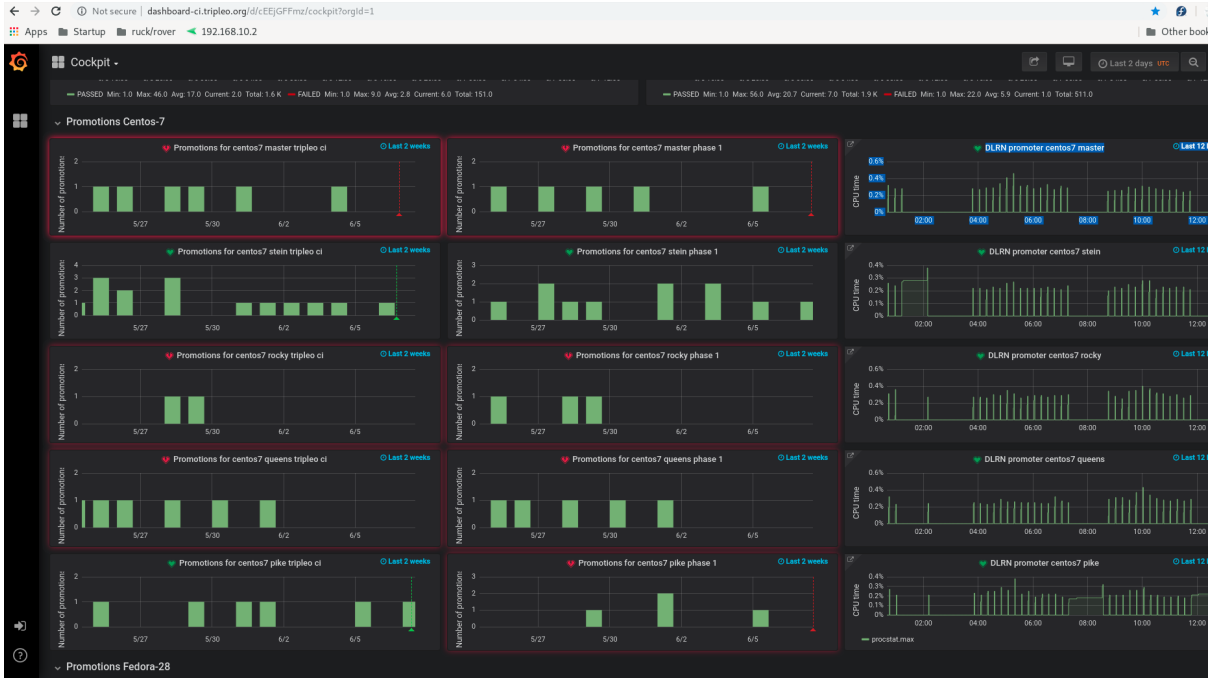


Fig. 2: As seen above on the left hand side and from top to bottom - the latest promotions for master, stein, rocky, queens and pike as bar charts. The bars represent promotions and height shows the number of promotions on that day.

4.1.10 Chasing CI promotions

The purpose of this document is to go into more detail about the TripleO promotion from the point of view of the ci-squad ruck|rover.

There is other documentation in this repo which covers the stages of the Tripleo-CI promotion pipeline in promotion-stages-overview and also about relevant tooling such as the dlrn-api-promoter.

Ensuring promotions are happening regularly (including for all current stable/ branches) is one of the biggest responsibilities of the ruck|rover. As explained in promotion-stages-overview the CI promotion represents the point at which we test all the tripleo-* things against the rest of openstack. The requirement is that there is a successful promotion (more on that below) at least once a week. Otherwise the branch will be considered in the red as in master promotion is red or we are red for stein promotion meaning was no promotion in (at least) 7 days for that branch.

Successful promotion

So what does it actually mean to have a successful promotion. In short:

- The TripleO periodic jobs have to run to completion and
- The periodic jobs in the promotion criteria must pass and
- The promoter server must be running in order to actually notice the job results and promote!

Each of these is explained in more detail below.

TripleO periodic jobs

The TripleO periodic jobs are ci jobs that are executed in one of the TripleO periodic pipelines. At time of writing we have four periodic pipelines defined in the config repo zuul pipelines:

```
* openstack-periodic-master
* openstack-periodic-latest-released
* openstack-periodic-24hr
* openstack-periodic-wednesday-weekend
```

These pipelines are *periodic* because unlike the check and gate pipelines (see ci jobs for more on those) jobs that run on each submitted code review, periodic jobs are executed *periodically*, at an interval given in cron syntax as you can see in the definitions at config repo zuul pipelines):

```
- pipeline:
  name: openstack-periodic-master
  post-review: true
  description: Jobs in this queue are triggered to run every few hours.
  manager: independent
  precedence: high
  trigger:
    timer:
      - time: '10 0,12,18 * * *'
```

As can be seen at time of writing the openstack-periodic-master jobs will run three times every day, at 10 minutes after midnight, noon and 6pm.

The four pipelines correspond to the four latest releases of OpenStack. The openstack-periodic-master runs jobs for master promotion, openstack-periodic-latest-released runs jobs for the latest stable branch promotion, openstack-periodic-24hr runs jobs for the stable branch before that and finally openstack-periodic-wednesday-weekend runs jobs for the stable branch before that.

You can see the full list of jobs that are executed in the pipelines in the rdo-infra periodic zuul layout.

It is important to finally highlight a common pattern in the pipeline layout. In each case the first job that must complete is the promote-consistent-to-tripleo-ci-testing which is where we take the latest consistent hash and mark it as tripleo-ci-testing to become our new candidate (see promotion-stages-overview) to be used by the rest of the jobs in our pipeline. You will note that this is the only job that doesnt have any dependency:

```
...
- periodic-tripleo-ci-rhel-8-ovb-3ctlr_1comp-featureset001-master:
    dependencies:
      - periodic-tripleo-rhel-8-buildimage-ironic-python-agent-master
      - periodic-tripleo-rhel-8-master-containers-build-push
- periodic-tripleo-centos-7-master-promote-consistent-to-tripleo-ci-testing
...
```

Then the containers and overcloud image build jobs must complete and only then we finally run the rest of the jobs. These ordering requirements are expressed using dependencies in the layout:

```
...
- periodic-tripleo-rhel-8-buildimage-overcloud-full-master:
```

(continues on next page)

(continued from previous page)

```
    dependencies:
      - periodic-tripleo-centos-7-master-promote-consistent-to-tripleo-ci-
↪testing
- periodic-tripleo-rhel-8-buildimage-ironic-python-agent-master:
    dependencies:
      - periodic-tripleo-centos-7-master-promote-consistent-to-tripleo-ci-
↪testing
- periodic-tripleo-ci-centos-7-ovb-1ctlr_1comp-featureset002-master-upload:
    dependencies:
      - periodic-tripleo-centos-7-master-containers-build-push
..
```

As can be seen above the build image jobs depend on the promote-consistent job and then everything else in the layout depends on the container build job.

Promotion Server and Criteria

The promotion server is maintained by the Tripleo-CI squad at a secret location (!) and it runs the code from the DLRN API Promoter as a service. In short, the job of this service is to fetch the latest hashes from the RDO delorean service and then query the state of the periodic jobs using that particular hash.

The main input to the promotion server is the configuration which defines the promotion criteria. This is the list of jobs that must pass so that we can declare a successful promotion:

```
[current-tripleo]
periodic-tripleo-centos-7-master-containers-build-push
periodic-tripleo-ci-centos-7-ovb-3ctlr_1comp-featureset001-master
periodic-tripleo-ci-centos-7-ovb-1ctlr_1comp-featureset002-master-upload
periodic-tripleo-ci-centos-7-multinode-1ctlr-featureset010-master
periodic-tripleo-ci-centos-7-scenario001-standalone-master
periodic-tripleo-ci-centos-7-scenario002-standalone-master
periodic-tripleo-ci-centos-7-scenario003-standalone-master
...
```

The promoter service queries the delorean service for the results of those jobs (for a given hash) and if they are all found to be in SUCCESS then the hash can be promoted to become the new current-tripleo.

It is a common practice for TripleO CI ruck or rover to check the indexed promoter service logs to see why a given promotion is not successful for example or when debugging issues with the promotion code itself.

Hack the promotion with testproject

Finally testproject and the ability to run individual periodic jobs on demand is an important part of the ruck|rover toolbox. In some cases you may want to run a job for verification of a given launchpad bug that affects periodic jobs.

However another important use is when the ruck|rover notice that one of the jobs in criteria failed on something they (now) know how to fix, or on some unrelated/transient issue. Instead of waiting another 6 or however many hours for the next periodic to run, you can try to run the job yourself using testproject. If the job is successful in testproject and it is the only job missing from criteria then posting the testproject review can also mean directly causing the promotion to happen.

You first need to checkout testproject:

```
git clone https://review.rdoproject.org/r/testproject
cd testproject
vim .zuul.layout
```

To post a testproject review you simply need to add a .zuul.layout_ file:

```
- project:
    check:
      jobs:
        - periodic-tripleo-centos-7-train-containers-build-push:
            vars:
              force_periodic: true
```

So the above would run the periodic-tripleo-centos-7-train-containers-build-push. Note the required *force_periodic* variable which causes the job to run as though it is in the periodic pipeline, rather than in the check pipeline which you will use in testproject.

An example is there and if you need to include a known fix you can simply have a Depends-On in the commit message.

Specifying a particular hash

Jobs in the periodic promotion pipelines are using the tripleo-ci-testing repo as described in the promotion-stages-overview, since that is the candidate we are trying to promote to current-tripleo. The tripleo-ci-testing and all other named tags in tripleo, are associated with a particular *hash* that identifies the delorean repo. For example looking at centos7 master tripleo-ci-testing at time of writing we see:

```
[delorean]
name=delorean-tripleo-ansible-544864ccc03b053317f5408b0c0349a42723ce73
baseurl=https://trunk.rdoproject.org/centos7/54/48/
↪544864ccc03b053317f5408b0c0349a42723ce73_ebb98bd9
enabled=1
gpgcheck=0
priority=1
```

So the centos7 master tripleo-ci-testing *hash* is *544864ccc03b053317f5408b0c0349a42723ce73_ebb98bd9a*. The corresponding repo is given by the baseurl above and if you navigate to that URL with your browser you can see the list of packages used in the jobs. Thus, the job specified in the example above

for testproject *periodic-tripleo-centos-7-train-containers-build-push* would use whatever the current tripleo-ci-testing points to.

However it is possible to override the particular hash (and thus repo) used by a job you run with testproject, using the dlrn_hash_tag featureset_override:

```
- project:
    check:
      jobs:
        - periodic-tripleo-ci-centos-7-ovb-1ctlr_1comp-featureset002-train-
↪upload:
            vars:
              force_periodic: true
              featureset_override:
                dlrn_hash_tag: 4b32d316befe0919fd98a147d84086bc0907677a_
↪046903a2
```

Thus, in the example above the periodic-tripleo-ci-centos-7-ovb-1ctlr_1comp-featureset002-train-upload job would run with the hash: *4b32d316befe0919fd98a147d84086bc0907677a_046903a2* regardless of the current value of tripleo-ci-testing.

The most common reason for overriding the hash in this way is when we notice that a particular job failed during one of the recent periodic pipeline runs. Looking at one of the indexed promoter service logs you may notice something like the following text:

```
2020-02-21 03:57:07,458 31360 INFO promoter Skipping promotion of centos7-
↪master
{'timestamp': 1582243926, 'distro_hash':
↪'ebb98bd9545e026f033683143ae39e9e236b3671',
'promote_name': 'tripleo-ci-testing', 'user': 'review_rdoproject_org',
'repo_url': 'https://trunk.rdoproject.org/centos7/54/48/
↪544864ccc03b053317f5408b0c0349a42723ce73_ebb98bd9',
'full_hash': '544864ccc03b053317f5408b0c0349a42723ce73_ebb98bd9',
'repo_hash': '544864ccc03b053317f5408b0c0349a42723ce73_ebb98bd9',
'commit_hash': '544864ccc03b053317f5408b0c0349a42723ce73'}
from tripleo-ci-testing to current-tripleo,
missing successful jobs: [u'periodic-tripleo-ci-centos-7-ovb-3ctlr_1comp-
↪featureset035-master',
u'periodic-tripleo-ci-centos-7-ovb-3ctlr_1comp-featureset001-master']
```

In particular note the last line missing successful jobs. This means that for the hash *544864ccc03b053317f5408b0c0349a42723ce73_ebb98bd9* a promotion could not happen, because in this particular run, those two identified jobs were failed. If the jobs were fixed in the meantime or you now know how to fix them and get a good result, you could re-run those with testproject specifying the particular hash. If they execute successfully then on the next run the promoter will promote that hash to become the new current-tripleo.

4.1.11 Gating github projects using TripleO CI jobs

In TripleO deployment, we consume OpenStack and non-openstack projects. In order to catch issues early, every patchset of the OpenStack projects is gated with TripleO CI jobs using Zuul.

With the help of an RDO software factory instance, we can also now gate non-openstack projects hosted on Github.

ceph-ansible and podman are the two non-openstack projects which are heavily used in TripleO deployments and are hosted on github and for which we have enabled TripleO CI jobs via github pull requests as described below.

Jobs running against ceph-ansible

ceph-ansible is used to deploy Ceph in standalone scenario 1 and 4 jobs. These jobs are defined in rdo-jobs repo.

On any ceph-ansible pull request, A user can trigger these jobs by leaving a comment with check-rdo on a pull request. It is currently done manually by the OpenStack developers.

Then, those jobs will appear in the RDO software factory Zuul status page under *github-check* pipeline.

On merged patches, periodic jobs are also triggered in *openstack-periodic-weekend* pipeline.

Jobs running against podman

In TripleO, OpenStack services are running in containers. The container lifecycle, healthcheck and execution is managed via systemd using paunch. Paunch under the hood uses podman.

The *podman* utility comes from libpod project.

Currently on each libpod pull request, tripleo ci based jobs get triggered automatically and get queued in *github-check* pipeline in RDO software factory Zuul instance.

TripleO jobs related to podman are defined in rdo-jobs-repo.

For gating libpod project, we run keystone based scenario000 minimal tripleo deployment job which tests the functionality of podman with keystone services. It takes 30 mins to finish the tripleo deployment.

Below is the example job definition for scenario000-job:

```
- job:
    name: tripleo-podman-integration-rhel-8-scenario000-standalone
    parent: tripleo-ci-base-standalone-periodic
    nodeset: single-rhel-8-node
    branches: ^master$
    run: playbooks/podman/install-podman-rpm.yaml
    required-projects:
      - name: github.com/containers/libpod
    vars:
      featureset: '052'
      release: master
      registry_login_enabled: false
      featureset_override:
```

(continues on next page)

(continued from previous page)

```
        standalone_environment_files:
          - 'environments/low-memory-usage.yaml'
          - 'ci/environments/scenario000-standalone.yaml'
          - 'environments/podman.yaml'
        run_tempest: false
        use_os_tempest: false
```

For re-running the tripleo jobs on libpod pull request, we can add *check-github* comment on the libpod pull requests itself.

On merged patches, periodic jobs also get triggerd in *openstack-regular* rdo-job-pipeline.

Report bugs when Jobs start failing

TripleO Jobs running against libpod and ceph-ansible projects might fail due to issue in libpod/ceph-ansible or in TripleO itself.

Once the status of any job is *FAILED* or *POST_FAILURE* or *RETRY_LIMIT*. Click on the job link and it will open the build result page. Then click on *log_url*, click on the *job-output.txt*. It contains the results of ansible playbook runs. Look for *ERROR* or failed messages. If looks something obvious. Please go ahead and create the bug on launchpad against tripleo project with all the information.

Once the bug is created, please add *depcheck* tag on the filed launchpad bug. This tag is explicitly used for listing bugs related to TripleO CI job failure against ceph-ansible and podman projects.

check-rdo vs *check-github*

check-rdo and *check-github* comments are used to trigger TripleO based zuul jobs against github projects (ceph-ansible/podman) s pull requests.

Note: On commenting *check-rdo* or *check-github*, not all jobs will appears in the github-manual pipeline. It depends whether the jobs are configured in the particular pipeline to get triggered. If the jobs are not defined there then, nothing will happen.

---

check-rdo

It is used against ceph-ansible pull requests especially. The jobs will be gets triggered and land in *github-check* pipeline.

check-github

If a TripleO job fails against ceph-ansible or podman PRs, then it can be relaunched using *check-github* comment. The job will appear in *github-manual* pipeline.

Using *Depends-On* on ceph-ansible/podman pull requests

One can also create/put OpenStack or RDO gerrit reviews against ceph-ansible/podman pull requests by putting *Depends-On: <openstack/rdo gerrit review link>* in the first message of the github pull request.

4.1.12 Content Provider Jobs

This section gives an overview and some details about the content provider zuul jobs. They are so called because they consist of a parent job that builds containers which are then consumed by any number of child jobs. Thus the parent jobs are the content provider for the child jobs.

Why Do We Need Content Providers?

The content provider jobs were added by the Tripleo CI squad during the Victoria development cycle. Prior to this check and gate tripleo-ci jobs running on review.opendev.org code submissions were pulling the promoted current-tripleo containers from docker.io.

Having all jobs pull directly from a remote registry obviously puts a strain on resources; consider multiple jobs per code submission with tens of container pulls for each. We have over time been affected by a number of issues related to the container pulls (such as timeouts) that would cause jobs to fail and block the gates. Furthermore, docker has recently announced that requests will be rate limited to one or two hundred pull requests per six hours (without and with authentication respectively) on the free plan effective 01 November 2020.

In anticipation of this the TripleO CI squad has moved all jobs to the new content provider architecture.

The Content Provider

The main task executed by the content provider job is to build the containers needed to deploy TripleO. This is achieved with a collection of ansible plays defined in the multinode-standalone-pre.yml tripleo-quickstart-extras playbook.

Once built, the content provider then needs to make those containers available for use by the child jobs. The build-container role itself as invoked in multinode-standalone-pre.yml ensures containers are pushed to the a local registry on the content provider node. However the child job will need to know the IP address on which they can reach that registry.

To achieve this we use the zuul_return module that allows for a parent job to return data for consumption within child jobs. We set the required zuul_return data in the run-provider.yml playbook:

```
- name: Set registry IP address
  zuul_return:
    data:
      zuul:
        pause: true
```

(continues on next page)

(continued from previous page)

```
      registry_ip_address: "{{ hostvars[groups.all[0]].ansible_host }}"
      provider_dlrn_hash: "{{ dlrn_hash|default('') }}"
      provider_dlrn_hash_tag: "{{ dlrn_hash_tag|default('') }}"
      provider_job_branch: "{{ provider_job_branch }}"
      registry_ip_address_branch: "{{ registry_ip_address_branch }}"
      provider_dlrn_hash_branch: "{{ provider_dlrn_hash_branch }}"
  tags:
    - skip_ansible_lint
```

Child jobs retrieve the IP address for the content provider container registry via the registry_ip_address_branch dictionary. This contains a mapping between the release (master, victoria, ussuri etc) and the IP address of the content provider container registry with images for that release. For example:

```
registry_ip_address_branch:
  master: 38.145.33.72
```

Most jobs will only ever have one release in this dictionary but upgrade jobs will require two (more on that later). Note that besides setting the zuul_return data the task above sets the zuul pause: true. As the name suggests, this allows the parent content provider job to be paused until all children have executed.

Given all the above, it should be of little surprise ;) that the content provider zuul job definition is as follows (at time of writing):

```
- job:
    name: tripleo-ci-centos-8-content-provider
    parent: tripleo-ci-base-standalone-centos-8
    branches: ^(?!stable/(newton|ocata|pike|queens|rocky|stein)).*$
    run:
      - playbooks/tripleo-ci/run-v3.yaml
      - playbooks/tripleo-ci/run-provider.yml
    vars:
      featureset: '052'
      provider_job: true
      build_container_images: true
      ib_create_web_repo: true
      playbooks:
        - quickstart.yml
        - multinode-standalone-pre.yml
```

It uses the same featureset as the standalone job. Notice the multinode-standalone-pre.yml passed to tripleo-quickstart for execution. The run-provider.yml playbook is executed as the last of the zuul *run* plays.

Finally, one other important task performed by the content provider job is to build any dependent changes (i.e. depends-on in the code submission). This is done with build-test-packages invoked in the multinode-standalone-pre.yml. We ensure that the built repo is available to child jobs by setting the ib_create_web_repo variable when built-test-packages is invoked by a provider job. This makes the repo available via a HTTP server on the content provider node that consumers then retrieve as described below.

The Content Consumers

The child jobs or content consumers must use the container registry available from the content provider. To do this we set the docker_registry_host variable using the *job.registry_ip_address_branch* zuul_data returned from the parent content provider.

Any dependent changes built by build-test-packages are installed into consumer jobs using the install-built-repo playbook. This has been added into the appropriate base job definitions as a *pre-run:* play.

Finally, in order to make a given zuul job a *consumer* job we must set the content provider as dependency and pass the relevant variables. For example in order to run tripleo-ci-centos-8-scenario001-standalone as a consumer job:

```
- tripleo-ci-centos-8-content-provider
- tripleo-ci-centos-8-scenario001-standalone:
    files: *scen1_files
    vars: &consumer_vars
      consumer_job: true
      build_container_images: false
      tags:
        - standalone
    dependencies:
      - tripleo-ci-centos-8-content-provider
```

Upgrade Jobs

Upgrade jobs are a special case because they require content from more than one release. For instance tripleo-ci-centos-8-standalone-upgrade-ussuri will deploy train containers and then upgrade to ussuri containers.

To achieve this we use two content provider jobs as dependencies for the upgrade jobs that require them (not all do):

```
- tripleo-ci-centos-8-standalone-upgrade:
    vars: *consumer_vars
    dependencies:
      - tripleo-ci-centos-8-content-provider
      - tripleo-ci-centos-8-content-provider-ussuri
```

As shown earlier in this document the registry_ip_address_branch dictionary maps release to the appropriate registry. This is set by each of the two parent jobs and once both have executed the dictionary will contain more than one entry. For example:

```
registry_ip_address_branch:
  master: 213.32.75.192
  ussuri: 158.69.75.154
```

The consumer upgrade jobs then use the appropriate registry for the deployment or upgrade part of the test.

4.1.13 TripleO Dependency Pipeline

This section introduces the TripleO Dependency Pipeline. The dependency pipeline is what the TripleO CI team calls the series of zuul CI jobs that aim to catch problems in deployment *dependencies*.

A dependency is any package that is not directly related to the deployment of OpenStack itself, such as OpenvSwitch, podman, buildah, pacemaker and ansible. Each time, these projects release a newer version, it breaks the OpenStack deployment and CI.

Currently we have promotion and component pipeline set up to detect OpenStack projects related issues early.

In order to detect the breakages from non-openstack projects, TripleO dependency pipeline has come into existence. Currently we have a single type of pipeline enabled:

- packages coming from specific repo

The configurations for each pipeline can be found under tripleo-quickstart/src/branch/master/config/release/dependency_ci/<module/repo name>/repo_config.yaml.

Current OpenStack Dependency Pipeline jobs

- openstack-dependencies-openvswitch - for testing OVS and OVN dependencies coming from NFV sig repo.
- openstack-dependencies-centos-compose - for testing jobs pinned to a specific CentOS compose build.

Note:

The following pipelines were deprecated in favor of CentOS Stream 9 adoption:

- openstack-dependencies-containertools - for testing container tools dependencies
- openstack-dependencies-centos8stream - for testing base operating system dependencies coming from CentOS-8 stream repo.

Understanding Package Dependency Pipeline

openstack-dependencies-openvswitch is a package dependency pipeline where we tests OVS and OVN packages coming from NFV sig.

Here is the config for the openvswitch dependency pipeline:

```
add_repos:
  - type: generic
    reponame: openvswitch-next
    filename: "openvswitch-next.repo"
    baseurl: "https://buildlogs.centos.org/centos/8/nfv/x86_64/openvswitch-2/"
    update_container: false
dependency_override_repos:
  - centos-nfv-openvswitch,http://mirror.centos.org/centos/8/nfv/x86_64/
↪openvswitch-2/
```

(continues on next page)

(continued from previous page)

```
dep_repo_cmd_after: |
  {% if dependency_override_repos is defined %}
  {% for item in dependency_override_repos %}
  sudo dnf config-manager --set-disabled {{ item.split(',')[0] }}
  {% endfor %}
  sudo dnf clean metadata;
  sudo dnf clean all;
  sudo dnf update -y;
  {% endif %}
```

What do the above terms mean? * *add_repos*: This is the test repo i.e. the one that is bringing us a newer than normal version of the package we are testing, OpenvSwitch in this case. * *dependency_override_repos*: It is used to disable or override a particular repo.

In the above case, openvswitch-next.repo repo will get generated due to repo setup and will disables the centos-nfv-openvswitch repo.

Before the deployment, *rdo-jobs/dependency/get-dependency-repo-content.yaml* playbook is used to set particular release file (in this case it is config/release/dependency_ci/openvswitch/repo_config.yaml) and then generate a diff of packages from dependency_override_repos and new repos added by add_repos option.

Below are the jobs running in openstack-dependencies-openvswitch pipeline on review.rdoproject.org.

```
openstack-dependencies-openvswitch:
  jobs:
    - periodic-tripleo-ci-centos-8-standalone-openvswitch-container-build-
↪master:
        dependencies:
          - periodic-tripleo-ci-centos-8-standalone-master
    - periodic-tripleo-ci-centos-8-scenario007-standalone-openvswitch-
↪container-build-master:
        dependencies:
          - periodic-tripleo-ci-centos-8-scenario007-standalone-master
    - periodic-tripleo-ci-centos-8-standalone-master:
        vars:
          force_periodic: false
    - periodic-tripleo-ci-centos-8-scenario007-standalone-master:
        vars:
          force_periodic: false
```

Understanding CentOS Compose Pinning Dependency

The dependency compose-repos works in the same line as package dependency jobs, with the difference that instead of setting up a single repository at a time, it consumes metadata from the provided compose URL and generates a set of repos as specified in the configuration snippet below:

```
...
add_repos:
  - type: compose_repos
```

(continues on next page)

(continued from previous page)

```
    compose_url: "https://odcs.stream.centos.org/production/latest-CentOS-
↪Stream/compose/"
    release: centos-stream-9
    disable_conflicting: true
    variants:
      - AppStream
      - BaseOS
      - HighAvailability
      - CRB
    disable_repos:
      - quickstart-centos-appstreams.repo
      - quickstart-centos-base.repo
      - quickstart-centos-highavailability.repo
      - quickstart-centos-crb.repo
```

The *compose_repos* repos type was created to generate a set of repos based on a compose repos URL and information about variants and conflicting repos. The *variants* will define which repos should be created from compose metadata, while *disable_conflicting* and *disable_repos* should guarantee that conflicting repos will be disabled in favor of the new ones. For more details on how repos are generated, please check yum-config-compose in setup-role and yum-config tool.

Note: The process of setting up compose-repos starts earlier in the job, before any call to *repo-setup*, in one of the pre playbooks defined in base jobs. You shall see the centos-compose-repos.yml playbook running in jobs that have *dependency* set to *centos-compose*, which sets up those repos using the same tools mentioned above. The purpose of the dependency config here is to keep those repos enabled when any other playbook or role calls *repo-setup*.

Testing Jobs Using Compose Pinning Dependency

In order to test any job against a CentOS compose build, which can be a compose newer or older than the available on CentOS mirrors, you will need to setup a new job definition and provide the following job variables:

```
- job:
  name: tripleo-ci-centos-9-standalone-compose-pinning
  parent: tripleo-ci-centos-9-standalone
  vars:
    dependency: centos-compose
    centos_compose_url: https://odcs.stream.centos.org/production/latest-
↪CentOS-Stream/compose/
    build_container_images: true
    containers_base_image: quay.io/centos/centos:stream9
```

- *dependency*: need to be set to *centos-compose*.
- *centos_compose_url*: CentOS compose URL to be tested. Note that the full URL ends with *compose*, because it is where compose *metadata* lives, required by *yum-config* tool to generate the repos. The default value is set to latest compose, which might be ahead of mirrors compose.

Note: In the example above, there is an enabled flag for *build_container_images*. It means that process of building containers will also use compose repositories.

Ensuring Correct Module or Repo is Used

Once a jobs runs and finishes in the dependency pipeline, we need to navigate to job log url. Under *logs/undercloud/home/zuul* directory, we can see two log files:

- control_repoquery_list.log.txt.gz - Contains a list of new packages coming from newly added repos.
- control_test_diff_table.log.txt.gz - contains a diff of the packages coming from new repo and overridden repo

All the above operation is done rdo-jobs/playbooks/dependency/diff-control-test.yaml playbook which uses compare_rpms project from ci-config/ci-scripts/infra-setup/roles/rrcockpit/files.

Note: The dependency *compose-repos* doesnt support rpm diff control test yet.

4.1.14 TripleO CI Zuul Jobs Parenting

When a developer submits a patch to TripleO repositories, their code is tested against a series of different TripleO CI jobs. Each job creates a different scenario for testing purposes.

The TripleO CI jobs are Zuul jobs, defined within TripleO projects under one of several locations: zuul.d directory, .zuul.yaml or zuul.yaml.

A Zuul job can be inherited in various child jobs as parent.

Zuul Job Parenting

In order to re-use a particular Zuul job, we create a set of standard base jobs, which contain ansible variables, required projects, pre-run, run, post-run steps and Zuul related variables.

These base job definitions are used as parent in various tripleo-ci jobs. The child job inherits attributes from the parent unless these are overridden by the child.

A child job can override the variable which is also defined in parent job.

TripleO CI Base jobs

TripleO CI base jobs are defined in zuul.d/base.yaml file in tripleo-ci repo.

Below is the list of base jobs and each is explained in a little more detail in subsequent sections:

- tripleo-ci-base-common-required-projects
- tripleo-ci-base-standard
- tripleo-ci-base-multinode-standard

- tripleo-ci-base-singlenode-standard
- tripleo-ci-base-standalone-standard
- tripleo-ci-base-standalone-upgrade-standard
- tripleo-ci-base-ovb-standard
- tripleo-ci-base-containers-standard
- tripleo-ci-base-images-standard
- tripleo-ci-content-provider-standard

tripleo-ci-base-common-required-projects

It contains a list of common required projects and ansible roles which are needed to start the deployment. It is used in upstream, RDO and Downstream. If a new project is needed in all types of deployment (upstream, RDO and Downstream) it can be added here.

tripleo-ci-base-standard

It contains a set of ansible variables and playbooks used in most deployments.

tripleo-ci-base-multinode-standard

It contains a set of ansible variables and playbooks used in most containers multinode and scenarios job.

It is used in those jobs where the user needs to deploy OpenStack using one undercloud and one controller.

tripleo-ci-base-singlenode-standard

It contains a set of ansible variables and playbooks used in most single node jobs.

It is used in those jobs where user needs to build containers and overcloud images which later can be used in another deployment.

It can also be used for undercloud deployment.

tripleo-ci-base-standalone-standard

It contains a set of ansible variables and playbooks used in vanilla standalone and standalone based scenario jobs.

The standalone job consists of single node overcloud deployment.

tripleo-ci-base-standalone-upgrade-standard

It contains a set of ansible variables and playbooks used in the standalone upgrade job.

The singlenode job consists of single node overcloud deployment where we upgrade a deployment from an older release to a newer one.

tripleo-ci-base-ovb-standard

It contains a set of ansible variables and playbooks used in the virtual baremetal deployment.

The ovb job consists of one undercloud and four overcloud nodes (one compute and multiple controllers) deployed as virtual baremetal nodes. It is a replica of real world customer deployments.

It is used in RDO and downstream jobs.

tripleo-ci-base-containers-standard

It contains a set of ansible variables and playbooks used during build containers and pushing it to specific registry.

tripleo-ci-base-images-standard

It contains a set of ansible variables and playbooks used during build overcloud images and pushing it to image server.

tripleo-ci-content-provider-standard

It contains a set of ansible variables and playbooks used for building containers and pushing them to a local registry. Depends-on patches are built into respective rpm packages via DLRN and served by a local yum repos.

The job is paused to serve container registry and yum repos which can be used later in dependent jobs.

Currently these jobs are running in Upstream and Downstream.

Required Project Jobs

It contains the list of required projects needed for specific type of deployment.

Upstream job tripleo-ci-build-containers-required-projects-upstream requires projects like ansible-role-container-registry, kolla, python-tripleoclient, tripleo-ansible to build containers.

In case of RDO tripleo-ci-build-containers-required-projects-rdo serves the same purpose.

Many Upstream OpenStack projects are forked downstream and have different branches.

To accommodate the downstream namespace and branches we use the downstream specific required project job (*required-projects-downstream*) as a base job with proper branches and override-checkout.

tripleo-ci-base-required-projects-multinode-internal job defined in the examples are perfect example for the same.

Below is one of the examples of container multinode required projects job.

Upstream

```
- job:
    name: tripleo-ci-base-required-projects-multinode-upstream
    description: |
                Base abstract job to add required-projects for Upstream␣
↪Multinode Jobs
    abstract: true
    parent: tripleo-ci-base-multinode-standard
    required-projects:
      - opendev.org/openstack/tripleo-ansible
      - opendev.org/openstack/tripleo-common
      - opendev.org/openstack/tripleo-operator-ansible
      - name: opendev.org/openstack/ansible-config_template
        override-checkout: master
```

RDO

```
- job:
    name: tripleo-ci-base-required-projects-multinode-rdo
    abstract: true
    description: |
        Base abstract job for multinode in RDO CI zuulv3 jobs
    parent: tripleo-ci-base-multinode-standard
    pre-run:
      - playbooks/tripleo-rdo-base/pre.yaml
      - playbooks/tripleo-rdo-base/container-login.yaml
    roles:
      - zuul: opendev.org/openstack/ansible-role-container-registry
      - zuul: opendev.org/openstack/tripleo-ansible
    required-projects:
      - opendev.org/openstack/ansible-role-container-registry
      - opendev.org/openstack/tripleo-ansible
    secrets:
      - rdo_registry
    vars:
      registry_login_enabled: true
```

Downstream

```
- job:
    name: tripleo-ci-base-required-projects-multinode-internal
    description: |
        Base abstract job to add required-projects for multinode downstream␣
↪job
    abstract: true
    override-checkout: <downstream branch name>
    parent: tripleo-ci-base-multinode-standard
    required-projects:
      - name: tripleo-ansible
```

(continues on next page)

(continued from previous page)

```
      branch: <downstream-branch>
    - ansible-config_template
    - tripleo-operator-ansible
    - rdo-jobs
    - tripleo-environments
  roles:
    - zuul: rdo-jobs
  pre-run:
    - playbooks/configure-mirrors.yaml
    - playbooks/tripleo-rdo-base/cert-install.yaml
    - playbooks/tripleo-rdo-base/pre-keys.yaml
  vars:
    mirror_locn: <downstream mirror address>
    featureset_override:
      artg_repos_dir: /home/zuul/src/<downstream-url>/openstack
```

Distribution Jobs

The TripleO deployment is supported on multiple distro versions. Here is the current supported matrix in RDO, Downstream and Upstream.

Release	CentOS/CentOS Stream Version	RHEL Version
Master	9-Stream	•
Wallaby	8-Stream, 9-Stream	8.x, 9
Victoria	8-Stream	•
Ussuri	8-Stream	•
Train	7, 8-Stream	8.x

Each of these distros have different settings which are used in deployment. Its easier to maintain separate variables based on distributions.

Below is an example of distro jobs for containers multinode at different levels.

Upstream Distro Jobs

```
- job:
    name: tripleo-ci-base-multinode
    abstract: true
    description: |
                Base abstract job for multinode TripleO CI C7 zuulv3 jobs
    parent: tripleo-ci-base-required-projects-multinode-upstream
    nodeset: two-centos-7-nodes
```

(continues on next page)

(continued from previous page)

```
- job:
    name: tripleo-ci-base-multinode-centos-8
    abstract: true
    description: |
                Base abstract job for multinode TripleO CI centos-8 zuulv3␣
↪jobs
    parent: tripleo-ci-base-required-projects-multinode-upstream
    nodeset: two-centos-8-nodes

- job:
    name: tripleo-ci-base-multinode-centos-9
    abstract: true
    description: |
                Base abstract job for multinode TripleO CI centos-9 zuulv3␣
↪jobs
    parent: tripleo-ci-base-required-projects-multinode-upstream
    nodeset: two-centos-9-nodes
```

RDO Distro Jobs

```
- job:
    name: tripleo-ci-base-multinode-periodic
    parent: tripleo-ci-base-multinode-rdo
    pre-run: playbooks/tripleo-ci-periodic-base/pre.yaml
    post-run: playbooks/tripleo-ci-periodic-base/post.yaml
    required-projects:
      - config
      - rdo-infra/ci-config
    roles:
      - zuul: rdo-infra/ci-config
    secrets:
      - dlrnapi

- job:
    name: tripleo-ci-base-multinode-periodic-centos-8
    parent: tripleo-ci-base-multinode-rdo-centos-8
    pre-run: playbooks/tripleo-ci-periodic-base/pre.yaml
    post-run: playbooks/tripleo-ci-periodic-base/post.yaml
    required-projects:
      - config
      - rdo-infra/ci-config
    roles:
      - zuul: rdo-infra/ci-config
    vars:
      promote_source: tripleo-ci-testing
    secrets:
      - dlrnapi

- job:
```

(continues on next page)

(continued from previous page)

```
    name: tripleo-ci-base-multinode-periodic-centos-9
    parent: tripleo-ci-base-multinode-rdo-centos-9
    pre-run: playbooks/tripleo-ci-periodic-base/pre.yaml
    post-run: playbooks/tripleo-ci-periodic-base/post.yaml
    required-projects:
      - config
      - rdo-infra/ci-config
    roles:
      - zuul: rdo-infra/ci-config
    vars:
      promote_source: tripleo-ci-testing
    secrets:
      - dlrnapi
```

Zuul Job Inheritance Order

Here is an example of Upstream inheritance of tripleo-ci-centos-9-containers-multinode job.:

```
tripleo-ci-base-common-required-projects
   |
   v
tripleo-ci-base-standard
   |
   v
tripleo-ci-base-multinode-standard
   |
   v
tripleo-ci-base-required-projects-multinode-upstream
   |
   v
tripleo-ci-base-multinode-centos-9
   |
   v
tripleo-ci-centos-9-containers-multinode
```

Here is the another example of RDO job periodic-tripleo-ci-centos-8-containers-multinode-master

```
tripleo-ci-base-multinode-standard
   |
   v
tripleo-ci-base-required-projects-multinode-rdo
   |
   v
tripleo-ci-base-multinode-rdo-centos-8
   |
   v
tripleo-ci-base-multinode-periodic-centos-8
   |
```

(continues on next page)

(continued from previous page)

```
    v
periodic-tripleo-ci-centos-8-containers-multinode-master
```

TripleO CI Zuul Job Repos

Below is the list of repos where tripleo-ci related Zuul jobs are defined.

Upstream

- tripleo-ci

RDO

- config: Jobs which needs secrets are defined here.
- rdo-jobs

FAQs regarding TripleO CI jobs

- If we have a new project, which needs to be tested at all places and installed from source but
  - cloned from upstream source, then it must be added under required-projects at tripleo-ci-base-common-required-projects job.
  - the project namespace is different in Upstream and downstream, then it must be added under required-projects at Downstream (tripleo-ci-base-required-projects-multinode-internal) or Upstream (tripleo-ci-base-required-projects-multinode-upstream) specific required-projects parent job.
  - if the project is only developed at downstream or RDO or Upstream, then it must be added under required project at downstream or RDO or Upstream required-projects parent job.
- In order to add support for new distros, please use required-projects job as a parent and then create distro version specific child job with required nodeset.
- If a project with different branch is re-added in child job required-projects, then the child job project will be used in the deployment.
- If a playbook (which calls another role, exists in different repo) is called at pre-run step in Zuul job, then role specific required projects and roles needs to be added at that job level. For example: In tripleo-ci-containers-rdo-upstream-pre job, ansible-role-container-registry and triple-ansible is needed for pre.yaml playbook. So both projects are added in roles and required-projects.
- If a job having pre/post run playbook needs zuul secrets and playbook depends on distros, then the job needs to be defined in config repo.
- We should not use branches attributes in Zuul Distro jobs or options jobs.

CHAPTER

FIVE

INSTALL GUIDE

5.1 TripleO Install Guide

5.1.1 TripleO Introduction

TripleO is an OpenStack Deployment & Management tool.

Architecture

With TripleO, you start by creating an **undercloud** (an actual operator facing deployment cloud) that will contain the necessary OpenStack components to deploy and manage an **overcloud** (an actual tenant facing workload cloud). The overcloud is the deployed solution and can represent a cloud for any purpose (e.g. production, staging, test, etc). The operator can choose any of available Overcloud Roles (controller, compute, etc.) they want to deploy to the environment.

Go to *TripleO Architecture* to learn more.

Components

TripleO is composed of set of official OpenStack components accompanied by few other open source plugins which increase TripleOs capabilities.

Go to *TripleO Components* to learn more.

Deployment Guide

See additional information about how to deploy TripleO in the Deploy Guide.

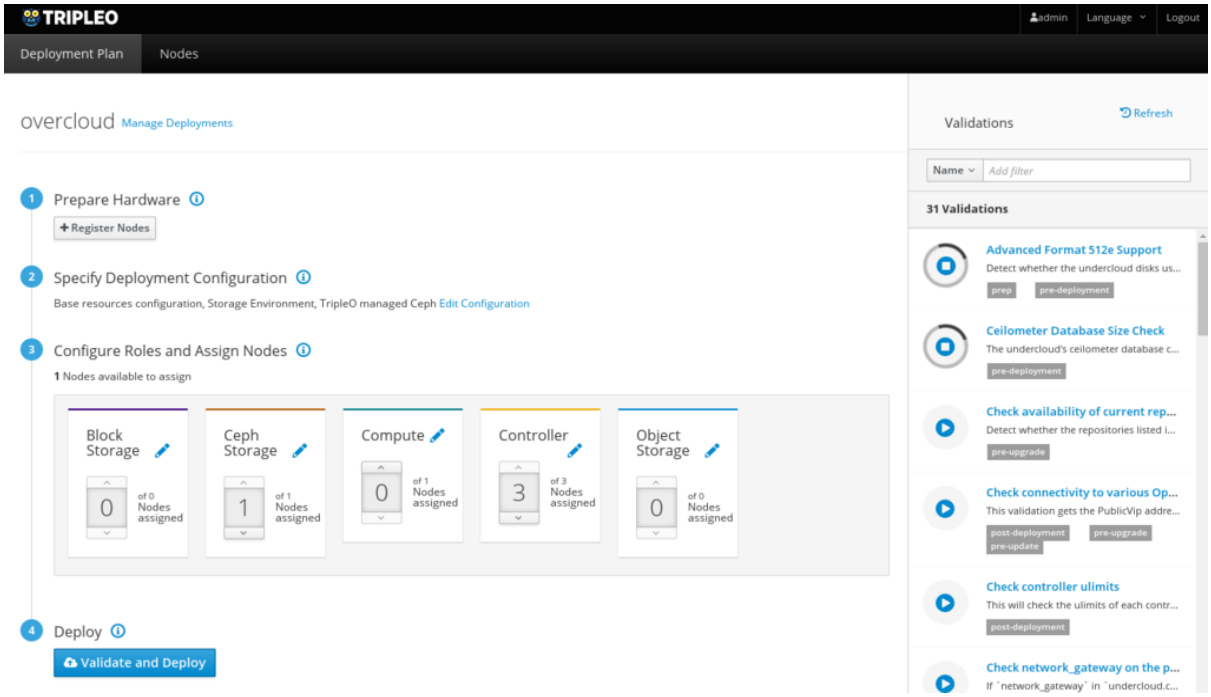
5.1.2 Deploy Guide

The installation instructions have been moved to the TripleO Deploy Guide.

5.1.3 (DEPRECATED) Basic Deployment (UI)

Note: The tripleo-ui is no longer available as of Stein. This documentation is deprecated.

This document will show you how to access the TripleO UI and perform a simple deployment with some customizations. Validations are automatically run at every step to help uncover potential issues early.



Prepare Your Environment

The UI is installed by default with the undercloud. You can confirm this by opening `undercloud.conf` and checking for:

```
enable_ui = true
enable_validations = true
```

The validations are optional but strongly recommended.

1. Make sure you have your environment ready and undercloud running:
   - Environment Setup
   - Undercloud Installation
1. Make sure the images are uploaded in Glance on the undercloud:
   - Get Images
   - Upload Images

Access the UI

The UI is accessible on the undercloud URL. With the default settings the URLs may look like the following, depending on whether the undercloud was set up with SSL:

- http://192.168.24.1:3000 if it was not
- https://192.168.24.2 if set up with SSL

The username is `admin` and the password can be obtained by running the following command on the undercloud:

```
$ sudo hiera keystone::admin_password
```

Note: On an undercloud deployed without SSL, the UI and API endpoints are deployed on the control plane which may not be routable. In this case you will need to create a tunnel or use a tool such as sshuttle to be able to use the UI from a local browser:

```
sshuttle -r user@undercloud 192.168.24.0/24
```

Virtual

If you cannot directly access the undercloud (for example because the undercloud is installed in a VM on a remote lab host), you will need to create a tunnel and make some configuration changes in order to access the UI locally.

1. Open the tunnel from the virt host, to the undercloud:

   ```
   ssh -Nf user@undercloud -L 0.0.0.0:443:192.168.24.2:443 # If SSL
   ssh -Nf user@undercloud -L 0.0.0.0:3000:192.168.24.1:3000 # If no SSL
   ```

   Note: Quickstart started creating the tunnel automatically during Pike. If using an older version you will have to create the tunnel manually, for example:

   ```
   ssh -F /root/.quickstart/ssh.config.ansible undercloud -L 0.0.0.0:443:192.
   ↪168.24.2:443
   ```

2. Edit the UI config on the undercloud to replace the undercloud IP with your virt host name, for example:

   ```
   sudo sed -i.bak s/192.168.24.2/virthost/ /var/www/openstack-tripleo-ui/
   ↪dist/tripleo_ui_config.js
   ```

   Additionally, make sure all the API endpoints are commented out in this file.

   Note: Changes to `tripleo_ui_config.js` are overwritten on undercloud upgrades.

3. You may have to open port 3000 or 443 on your virt host.

Stable Branch

Newton

Starting in Ocata, all the API ports are proxied through 3000 (non-SSL) or 443 (SSL). If using Newton, you will need to ensure ports for all the API endpoints specified in `tripleo_ui_config.js` are open and accessible. If using SSL with self-signed certificates, Firefox will also require a SSL certificate exception to be accepted for every port.

4. The UI should now be accessible at http://virthost:3000 or https://virthost.

Manage Plans

A default plan named `overcloud` is created during the undercloud installation, based on the default tripleo-heat-templates installed on the system. This plan can be customized and deployed.

It is also possible to create and manage additional plans in parallel, in order to test different configurations.

By clicking on Manage Deployments beside the deployment name, you can perform actions on plans such as create, export, delete, etc.

Note: There can be confusion with node assignments when switching between plans, particularly in previous releases like Newton. If doing work with multiple plans, ensure the Node counts are what you expect before starting the deployment, for example by navigating to Edit Configuration -> Parameters.

Manage Nodes

Register Nodes

Navigate to the **Nodes** tab in the top bar and click on the *Register Nodes* button. New nodes can be added in two ways:

- Importing an instackenv.json file
- Importing an instackenv.json file
- Manually defining nodes via the *Add New* button.

Introspect Nodes

Introspection is a required step when deploying from the UI. Once the nodes are registered and in the `manageable` provision state, select the nodes and click on the *Introspect Nodes* button.

Provide Nodes

Once introspection is completed, nodes need to be provided in order to move to the `available` state and be available for deployments. Select the nodes and click on the *Provide Nodes* button.

Note: For more information about node states, see Node States.

Tag Nodes

Nodes need to be tagged to match a specific profile/role before they can be used in a deployment. Select the nodes you want to assign a profile to, then click on *Tag Nodes* (the option may be in a dropdown menu).

Stable Branch

In Newton and Ocata, node assignment and node tagging are done at the same time when assigning nodes on the **Deployment Plan** page.

Configure the Deployment

On the **Deployment Plan** tab, you can configure both the overall deployment, as well as specific roles.

Clicking on the *Edit Configuration* link displays the list of environments available and their description, based on the templates provided in the plan. After enabling environments as desired, click on *Save Changes* and navigate to the **Parameters** tab. Once saved, the enabled environments will also be configurable on this tab.

The **Parameters** tab lets you set configuration options for the deployment in general, as well as for each individual environment.

Stable Branch

Newton

In Newton it was not possible to configure individual environments. The environment templates should be updated directly with the required parameters before uploading a new plan.

Individual roles can also be configured by clicking on the Pencil icon beside the role name on each card.

Stable Branch

Newton

In Newton, you may need to assign at least one node to the role before the related configuration options are loaded.

Assign Nodes

The second number on each card indicates the number of nodes tagged with this particular profile. The number of nodes manually assigned via the number picker will be deployed.

Stable Branch

In Newton and Ocata, nodes are tagged as part of assigning a node to a profile. This can cause issues when switching deployment plans, as the node counts displayed on the card may not match the value actually stored in the plan. You can correct this by clicking on Edit Configuration -> Parameters and checking/updating the node counts for ControllerCount, ComputeCount, etc.

Additionally, when using custom roles you should make sure to unassign the nodes associated with these roles before deleting the plan, as the role cards are displayed based on the roles in the current plan only. Therefore it is not possible to unassign a node which is associated with a role that does not exist in the current plan.

Deploy the Overcloud

Click on the *Deploy* button.

You may see a warning if not all validations passed. While this is expected in resources-constrained virtual environments, it is recommended to check the failed validations carefully before proceeding.

The `View detailed information` link shows the details for all the Heat resources being deployed.

Post-Deployment

Once the deployment completes, the progress bar will be replaced with information about the overcloud such as the IP address and login credentials.

If the deployment failed, information about the failure will be displayed.

Virtual

To access the overcloud, you will need to update your tunnel in order to access the new URL. For example, if your overcloud information is as such:

```
Overcloud IP address: 192.168.24.12
Username: admin
Password: zzzzzz
```

Assuming you deployed the overcloud with SSL enabled, you could create the following tunnel from your virt host to the undercloud:

```
ssh -Nf user@undercloud -L 0.0.0.0:1234:192.168.24.12:443
```

After opening port 1234 on your virt host, you should be able to access the overcloud by navigating to https://virthost:1234.

5.1.4 Feature Configuration

Documentation on how to enable and configure various features available in TripleO.

(DEPRECATED) Deploying OpenShift

Note: This functionality was removed as of Train.

You can use TripleO to deploy OpenShift clusters onto baremetal nodes. TripleO deploys the operating system onto the nodes and uses *openshift-ansible* to then configure OpenShift. TripleO can also be used to manage the baremetal nodes.

Define the OpenShift roles

TripleO installs OpenShift services using composable roles for *OpenShiftMaster*, *OpenShiftWorker*, and *OpenShiftInfra*. When you import a baremetal node using *instackenv.json*, you can tag it to use a certain composable role. See Custom Roles for more information.

1. Generate the OpenShift roles:

```
openstack overcloud roles generate -o /home/stack/openshift_roles_data.yaml \
  OpenShiftMaster OpenShiftWorker OpenShiftInfra
```

2. View the OpenShift roles:

```
openstack overcloud role list
```

The result should include entries for *OpenShiftMaster*, *OpenShiftWorker*, and *OpenShiftInfra*.

3. See more information on the *OpenShiftMaster* role:

```
openstack overcloud role show OpenShiftMaster
```

Note: For development or PoC environments that are more resource-constrained, it is possible to use the *OpenShiftAllInOne* role to collocate the different OpenShift services on the same node. The all-in-one role is not recommended for production.

Create the OpenShift profiles

This procedure describes how to enroll a physical node as an OpenShift node.

1. Create a flavor for each OpenShift role. You will need to adjust this values to suit your requirements:

```
openstack flavor create --id auto --ram 4096 --disk 40 --vcpus 1 --swap 500␣
↪m1.OpenShiftMaster
openstack flavor create --id auto --ram 4096 --disk 40 --vcpus 1 --swap 500␣
↪m1.OpenShiftWorker
openstack flavor create --id auto --ram 4096 --disk 40 --vcpus 1 --swap 500␣
↪m1.OpenShiftInfra
```

2. Map the flavors to the required profile:

```
openstack flavor set --property "capabilities:profile"="OpenShiftMaster" \
  --property "capabilities:boot_option"="local" m1.OpenShiftMaster
openstack flavor set --property "capabilities:profile"="OpenShiftWorker" \
  --property "capabilities:boot_option"="local" m1.OpenShiftWorker
openstack flavor set --property "capabilities:profile"="OpenShiftInfra" \
  --property "capabilities:boot_option"="local" m1.OpenShiftInfra
```

3. Add your nodes to *instackenv.json*. You will need to define them to use the *capabilities* field. For example:

```
[{
    "arch":"x86_64",
    "cpu":"4",
    "disk":"60",
    "mac":[
            "00:0c:29:9f:5f:05"
    ],
    "memory":"16384",
    "pm_type":"ipmi",
    "capabilities":"profile:OpenShiftMaster",
    "name": "OpenShiftMaster_1"
},
{
    "arch":"x86_64",
    "cpu":"4",
    "disk":"60",
    "mac":[
            "00:0c:29:91:b9:2d"
    ],
    "memory":"16384",
    "pm_type":"ipmi",
    "capabilities":"profile:OpenShiftWorker",
    "name": "OpenShiftWorker_1"
},
{
    "arch":"x86_64",
```

(continues on next page)

(continued from previous page)

```
    "cpu":"4",
    "disk":"60",
    "mac":[
            "00:0c:29:91:b9:6a"
    ],
    "memory":"16384",
    "pm_type":"ipmi",
    "capabilities":"profile:OpenShiftInfra",
    "name": "OpenShiftInfra_1"
}]
```

4. Import and introspect the TripleO nodes as you normally would for your deployment. For example:

```
openstack overcloud node import ~/instackenv.json
openstack overcloud node introspect --all-manageable --provide
```

5. Verify the overcloud nodes have assigned the correct profile

```
openstack overcloud profiles list
+--------------------------------------+--------------------+-----------------
↪+-----------------+-------------------+
| Node UUID                            | Node Name          | Provision State␣
↪| Current Profile | Possible Profiles |
+--------------------------------------+--------------------+-----------------
↪+-----------------+-------------------+
| 72b2b1fc-6ba4-4779-aac8-cc47f126424d | openshift-worker01 | available       ␣
↪| OpenShiftWorker |                   |
| d64dc690-a84d-42dd-a88d-2c588d2ee67f | openshift-worker02 | available       ␣
↪| OpenShiftWorker |                   |
| 74d2fd8b-a336-40bb-97a1-adda531286d9 | openshift-worker03 | available       ␣
↪| OpenShiftWorker |                   |
| 0eb17ec6-4e5d-4776-a080-ca2fdcd38e37 | openshift-infra02  | available       ␣
↪| OpenShiftInfra  |                   |
| 92603094-ba7c-4294-a6ac-81f8271ce83e | openshift-infra03  | available       ␣
↪| OpenShiftInfra  |                   |
| b925469f-72ec-45fb-a403-b7debfcf59d3 | openshift-master01 | available       ␣
↪| OpenShiftMaster |                   |
| 7e9e80f4-ad65-46e1-b6b4-4cbfa2eb7ea7 | openshift-master02 | available       ␣
↪| OpenShiftMaster |                   |
| c2bcdd3f-38c3-491b-b971-134cab9c4171 | openshift-master03 | available       ␣
↪| OpenShiftMaster |                   |
| ece0ef2f-6cc8-4912-bc00-ffb3561e0e00 | openshift-infra01  | available       ␣
↪| OpenShiftInfra  |                   |
| d3a17110-88cf-4930-ad9a-2b955477aa6c | openshift-custom01 | available       ␣
↪| None            |                   |
| 07041e7f-a101-4edb-bae1-06d9964fc215 | openshift-custom02 | available       ␣
↪| None            |                   |
+--------------------------------------+--------------------+-----------------
↪+-----------------+-------------------+
```

Configure the container registry

Follow container image preparation to configure TripleO for the container image preparation.

This generally means generating a */home/stack/containers-prepare-parameter.yaml* file:

```
openstack tripleo container image prepare default \
  --local-push-destination \
  --output-env-file containers-prepare-parameter.yaml
```

Define the OpenShift environment

Create the *openshift_env.yaml* file. This file will define the OpenShift-related settings that TripleO will later apply as part of the *openstack overcloud deploy* procedure. You will need to update these values to suit your deployment:

```
Parameter_defaults:
# by default TripleO assigns the VIP random from the allocation pool
# by using the FixedIPs we can set the VIPs to predictable IPs before␣
↪starting the deployment
CloudName: public.openshift.localdomain
PublicVirtualFixedIPs: [{'ip_address':'10.0.0.200'}]

CloudNameInternal: internal.openshift.localdomain
InternalApiVirtualFixedIPs: [{'ip_address':'172.17.1.200'}]

CloudDomain: openshift.localdomain

## Required for CNS deployments only
OpenShiftInfraParameters:
    OpenShiftGlusterDisks:
        - /dev/sdb

## Required for CNS deployments only
OpenShiftWorkerParameters:
    OpenShiftGlusterDisks:
        - /dev/sdb
        - /dev/sdc

ControlPlaneDefaultRoute: 192.168.24.1
EC2MetadataIp: 192.168.24.1
ControlPlaneSubnetCidr: 24

# The DNS server below should have entries for resolving
# {internal,public,apps}.openshift.localdomain names
DnsServers:
    - 10.0.0.90

OpenShiftGlobalVariables:
```

(continues on next page)

(continued from previous page)

```
    openshift_master_identity_providers:
        - name: 'htpasswd_auth'
          login: 'true'
          challenge: 'true'
          kind: 'HTPasswdPasswordIdentityProvider'
    openshift_master_htpasswd_users:
        sysadmin: '$apr1$jpBOUqeU$X4jUsMyCHOOp8TFYtPq0v1'

    #openshift_master_cluster_hostname should match the CloudNameInternal␣
↪parameter
    openshift_master_cluster_hostname: internal.openshift.localdomain

    #openshift_master_cluster_public_hostname should match the CloudName␣
↪parameter
    openshift_master_cluster_public_hostname: public.openshift.localdomain

    openshift_master_default_subdomain: apps.openshift.localdomain
```

For custom networks or customer interfaces, it is necessary to use custom network interface templates:

```
resource_registry:
    OS::TripleO::OpenShiftMaster::Net::SoftwareConfig: /home/stack/master-nic.
↪yaml
    OS::TripleO::OpenShiftWorker::Net::SoftwareConfig: /home/stack/worker-nic.
↪yaml
    OS::TripleO::OpenShiftInfra::Net::SoftwareConfig: /home/stack/infra-nic.
↪yaml
```

Deploy OpenShift nodes

As a result of the previous steps, you will have three new YAML files:

- *openshift_env.yaml*
- *openshift_roles_data.yaml*
- *containers-default-parameters.yaml*

For a custom network deployments, maybe it is necessary NICs and network templates like:

- *master-nic.yaml*
- *infra-nic.yaml*
- *worker-nic.yaml*
- *network_data_openshift.yaml*

Add these YAML files to your *openstack overcloud deploy* command.

An example for CNS deployments:

```
openstack overcloud deploy \
  --stack openshift \
  --templates \
  -r /home/stack/openshift_roles_data.yaml \
  -n /usr/share/openstack-tripleo-heat-templates/network_data_openshift.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/network-
↪isolation.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/openshift.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/openshift-cns.
↪yaml \
  -e /home/stack/openshift_env.yaml \
  -e /home/stack/containers-prepare-parameter.yaml
```

An example for non-CNS deployments:

```
openstack overcloud deploy \
  --stack openshift \
  --templates \
  -r /home/stack/openshift_roles_data.yaml \
  -n /usr/share/openstack-tripleo-heat-templates/network_data_openshift.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/network-
↪isolation.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/openshift.yaml \
  -e /home/stack/openshift_env.yaml \
  -e /home/stack/containers-prepare-parameter.yaml
```

Deployment for custom networks or interfaces, it is necessary to specify them. For example:

```
openstack overcloud deploy \
  --stack openshift \
  --templates \
  -r /home/stack/openshift_roles_data.yaml \
  -n /home/stack/network_data_openshift.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/network-
↪isolation.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/openshift.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/openshift-cns.
↪yaml \
  -e /home/stack/openshift_env.yaml \
  -e /home/stack/containers-prepare-parameter.yaml \
  -e /home/stack/custom-nics.yaml
```

Review the OpenShift deployment

Once the overcloud deploy procedure has completed, you can review the state of your OpenShift nodes.

1. List all your baremetal nodes. You should expect to see your master, infra, and worker nodes.

```
baremetal node list
```

2. Locate the OpenShift node:

```
openstack server list
```

3. SSH to the OpenShift node. For example:

```
ssh heat-admin@192.168.122.43
```

4. Change to root user:

```
sudo -i
```

5. Review the container orchestration configuration:

```
cat .kube/config
```

6. Login to OpenShift:

```
oc login -u admin
```

7. Review any existing projects:

```
oc get projects
```

8. Review the OpenShift status:

```
oc status
```

9. Logout from OpenShift:

```
oc logout
```

Deploy a test app using OpenShift

This procedure describes how to create a test application in your new OpenShift deployment.

1. Login as a developer:

```
$ oc login -u developer
Logged into "https://192.168.64.3:8443" as "developer" using existing␣
↪credentials.
You have one project on this server: "myproject"
Using project "myproject".
```

2. Create a new project:

```
$ oc new-project test-project
Now using project "test-project" on server "https://192.168.64.3:8443".
```

You can add applications to this project with the new-app command. For example, to build a new example application in Ruby try:

```
$ oc new-app centos/ruby-22-centos7~https://github.com/openshift/ruby-ex.
↪git
```

3. Create a new app. This example creates a CakePHP application:

```
$ oc new-app https://github.com/sclorg/cakephp-ex
--> Found image 9dd8c80 (29 hours old) in image stream "openshift/php"␣
↪under tag "7.1" for "php"

    Apache 2.4 with PHP 7.1
    -----------------------
    PHP 7.1 available as container is a base platform for building and␣
↪running various PHP 7.1 applications and frameworks. PHP is an HTML-
↪embedded scripting language. PHP attempts to make it easy for␣
↪developers to write dynamically generated web pages. PHP also offers␣
↪built-in database integration for several commercial and non-commercial␣
↪database management systems, so writing a database-enabled webpage with␣
↪PHP is fairly simple. The most common use of PHP coding is probably as␣
↪a replacement for CGI scripts.

    Tags: builder, php, php71, rh-php71

    * The source repository appears to match: php
    * A source build using source code from https://github.com/sclorg/
↪cakephp-ex will be created
    * The resulting image will be pushed to image stream "cakephp-
↪ex:latest"
    * Use 'start-build' to trigger a new build
    * This image will be deployed in deployment config "cakephp-ex"
    * Ports 8080/tcp, 8443/tcp will be load balanced by service "cakephp-
↪ex"
    * Other containers can access this service through the hostname
↪"cakephp-ex"

--> Creating resources ...
    imagestream "cakephp-ex" created
    buildconfig "cakephp-ex" created
    deploymentconfig "cakephp-ex" created
    service "cakephp-ex" created
--> Success
    Build scheduled, use 'oc logs -f bc/cakephp-ex' to track its progress.
    Application is not exposed. You can expose services to the outside␣
↪world by executing one or more of the commands below:
    'oc expose svc/cakephp-ex'
    Run 'oc status' to view your app.
```

4. Review the new app:

```
$ oc status --suggest
In project test-project on server https://192.168.64.3:8443

svc/cakephp-ex - 172.30.171.214 ports 8080, 8443
dc/cakephp-ex deploys istag/cakephp-ex:latest <-
    bc/cakephp-ex source builds https://github.com/sclorg/cakephp-ex on␣
↪openshift/php:7.1
    build #1 running for 52 seconds - e0f0247: Merge pull request #105␣
↪from jeffdyoung/ppc64le (Honza Horak <hhorak@redhat.com>)
    deployment #1 waiting on image or update

Info:
* dc/cakephp-ex has no readiness probe to verify pods are ready to accept␣
↪traffic or ensure deployment is successful.
    try: oc set probe dc/cakephp-ex --readiness ...
* dc/cakephp-ex has no liveness probe to verify pods are still running.
    try: oc set probe dc/cakephp-ex --liveness ...

View details with 'oc describe <resource>/<name>' or list everything with
↪'oc get all'.
```

5. Review the pods:

```
$ oc get pods
NAME                 READY     STATUS    RESTARTS   AGE
cakephp-ex-1-build   1/1       Running   0          1m
```

6. Logout from OpenShift:

```
$ oc logout
```

5.1.5 Custom Configurations

Documentation on how to deploy custom configurations with TripleO.

CHAPTER

SIX

UPGRADES/UPDATES/FFWD-UPGRADE

6.1 Upgrade, Update, FFWD Upgrade Guide

CHAPTER

SEVEN

DOCUMENTATION CONVENTIONS

Some steps in the following instructions only apply to certain environments, such as deployments to real baremetal and deployments using Red Hat Enterprise Linux (RHEL). These steps are marked as follows:

RHEL

Step that should only be run when using RHEL

RHEL Portal Registration

Step that should only be run when using RHEL Portal Registration

RHEL Satellite Registration

Step that should only be run when using RHEL Satellite Registration

CentOS

Step that should only be run when using CentOS

Baremetal

Step that should only be run when deploying to baremetal

Virtual

Step that should only be run when deploying to virtual machines

Ceph

Step that should only be run when deploying Ceph for use by the Overcloud

Stable Branch

Step that should only be run when choosing to use components from their stable branches rather than using packages/source based on current master.

Yoga

Step that should only be run when installing from the Yoga stable branch.

Zed

Step that should only be run when installing from the Zed stable branch.

2023.1 Antelope (SLURP)

Step that should only be run when installing from the 2023.1 Antelope (SLURP) stable branch.

2023.2 Bobcat

Step that should only be run when installing from the 2023.2 Bobcat stable branch.

Validations

Steps that will run the pre and post-deployment validations

Optional Feature

Step that is optional. A deployment can be done without these steps, but they may provide useful additional functionality.

Any such steps should *not* be run if the target environment does not match the section marking.