Red Hat OpenStack Platform 15-Beta

Advanced Overcloud Customization

Methods for configuring advanced features using Red Hat OpenStack Platform director

Last Updated: 2019-07-18

Red Hat OpenStack Platform 15-Beta Advanced Overcloud Customization

Methods for configuring advanced features using Red Hat OpenStack Platform director

OpenStack Team
rhos-docs@redhat.com

Legal Notice

Abstract

This guide explains how to configure certain advanced features for a Red Hat OpenStack Platform enterprise environment using the Red Hat OpenStack Platform Director. This includes features such as network isolation, storage configuration, SSL communication, and general configuration methods.

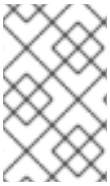
Table of Contents

CHAPTER 1. INTRODUCTION

The Red Hat OpenStack Platform director provides a set of tools to provision and create a fully featured OpenStack environment, also known as the Overcloud. The Director Installation and Usage Guide covers the preparation and configuration of the Overcloud. However, a proper production-level Overcloud might require additional configuration, including:

- Basic network configuration to integrate the Overcloud into your existing network infrastructure.
- Network traffic isolation on separate VLANs for certain OpenStack network traffic types.
- SSL configuration to secure communication on public endpoints
- Storage options such as NFS, iSCSI, Red Hat Ceph Storage, and multiple third-party storage devices.
- Registration of nodes to the Red Hat Content Delivery Network or your internal Red Hat Satellite 5 or 6 server.
- Various system-level options.
- Various OpenStack service options.

This guide provides instructions for augmenting your Overcloud through the director. At this point, the director has registered the nodes and configured the necessary services for Overcloud creation. Now you can customize your Overcloud using the methods in this guide.



NOTE

The examples in this guide are optional steps for configuring the Overcloud. These steps are only required to provide the Overcloud with additional functionality. Use the steps that apply to the needs of your environment.

CHAPTER 2. UNDERSTANDING HEAT TEMPLATES

The custom configurations in this guide use Heat templates and environment files to define certain aspects of the Overcloud. This chapter provides a basic introduction to Heat templates so that you can understand the structure and format of these templates in the context of the Red Hat OpenStack Platform director.

2.1. HEAT TEMPLATES

The director uses Heat Orchestration Templates (HOT) as a template format for its Overcloud deployment plan. Templates in HOT format are usually expressed in YAML format. The purpose of a template is to define and create a *stack*, which is a collection of resources that Heat creates, and the configuration of the resources. Resources are objects in OpenStack and can include compute resources, network configuration, security groups, scaling rules, and custom resources.

The structure of a Heat template has three main sections:

Parameters

These are settings passed to Heat, which provide a way to customize a stack, and any default values for parameters without passed values. These settings are defined in the **parameters** section of a template.

Resources

These are the specific objects to create and configure as part of a stack. OpenStack contains a set of core resources that span across all components. These are defined in the **resources** section of a template.

Output

These are values passed from Heat after the creation of the stack. You can access these values either through the Heat API or client tools. These are defined in the **output** section of a template.

Here is an example of a basic Heat template:

```
heat_template_version: 2013-05-23

description: > A very basic Heat template.

parameters:
  key_name:
    type: string
    default: lars
    description: Name of an existing key pair to use for the instance
  flavor:
    type: string
    description: Instance type for the instance to be created
    default: m1.small
  image:
    type: string
    default: cirros
    description: ID or name of the image to use for the instance

resources:
  my_instance:
    type: OS::Nova::Server
    properties:
```

```
      name: My Cirros Instance
      image: { get_param: image }
      flavor: { get_param: flavor }
      key_name: { get_param: key_name }

output:
  instance_name:
    description: Get the instance's name
    value: { get_attr: [ my_instance, name ] }
```

This template uses the resource type **type: OS::Nova::Server** to create an instance called **my_instance** with a particular flavor, image, and key. The stack can return the value of **instance_name**, which is called **My Cirros Instance**.

When Heat processes a template it creates a stack for the template and a set of child stacks for resource templates. This creates a hierarchy of stacks that descend from the main stack you define with your template. You can view the stack hierarchy using this following command:

```
$ openstack stack list --nested
```

2.2. ENVIRONMENT FILES

An environment file is a special type of template that provides customization for your Heat templates. This includes three key parts:

Resource Registry

This section defines custom resource names, linked to other Heat templates. This provides a method to create custom resources that do not exist within the core resource collection. These are defined in the **resource_registry** section of an environment file.

Parameters

These are common settings you apply to the top-level template's parameters. For example, if you have a template that deploys nested stacks, such as resource registry mappings, the parameters only apply to the top-level template and not templates for the nested resources. Parameters are defined in the **parameters** section of an environment file.

Parameter Defaults

These parameters modify the default values for parameters in all templates. For example, if you have a Heat template that deploys nested stacks, such as resource registry mappings,the parameter defaults apply to all templates. The parameter defaults are defined in the **parameter_defaults** section of an environment file.

IMPORTANT

It is recommended to use **parameter_defaults** instead of **parameters** When creating custom environment files for your Overcloud. This is so the parameters apply to all stack templates for the Overcloud.

An example of a basic environment file:

```
resource_registry:
  OS::Nova::Server::MyServer: myserver.yaml

parameter_defaults:
```

```
  NetworkName: my_network

parameters:
  MyIP: 192.168.0.1
```

For example, this environment file (**my_env.yaml**) might be included when creating a stack from a certain Heat template (**my_template.yaml**). The **my_env.yaml** files creates a new resource type called **OS::Nova::Server::MyServer**. The **myserver.yaml** file is a Heat template file that provides an implementation for this resource type that overrides any built-in ones. You can include the **OS::Nova::Server::MyServer** resource in your **my_template.yaml** file.

The **MyIP** applies a parameter only to the main Heat template that deploys along with this environment file. In this example, it only applies to the parameters in **my_template.yaml**.

The **NetworkName** applies to both the main Heat template (in this example, **my_template.yaml**) and the templates associated with resources included the main template, such as the **OS::Nova::Server::MyServer** resource and its **myserver.yaml** template in this example.

2.3. CORE OVERCLOUD HEAT TEMPLATES

The director contains a core Heat template collection for the Overcloud. This collection is stored in **/usr/share/openstack-tripleo-heat-templates**.

There are many Heat templates and environment files in this collection. However, the main files and directories to note in this template collection are:

overcloud.j2.yaml

This is the main template file used to create the Overcloud environment. This file uses Jinja2 syntax to iterate over certain sections in the template to create custom roles. The Jinja2 formatting is rendered into YAML during the Overcloud deployment process.

overcloud-resource-registry-puppet.j2.yaml

This is the main environment file used to create the Overcloud environment. It provides a set of configurations for Puppet modules stored on the Overcloud image. After the director writes the Overcloud image to each node, Heat starts the Puppet configuration for each node using the resources registered in this environment file. This file uses Jinja2 syntax to iterate over certain sections in the template to create custom roles. The Jinja2 formatting is rendered into YAML during the overcloud deployment process.

roles_data.yaml

A file that defines the roles in an overcloud and maps services to each role.

network_data.yaml

A file that defines the networks in an overcloud and their properties such as subnets, allocation pools, and VIP status. The default **network_data** file contains the default networks: External, Internal Api, Storage, Storage Management, Tenant, and Management. You can create a custom **network_data** file and add it to your **openstack overcloud deploy** command with the **-n** option.

plan-environment.yaml

A file that defines the metadata for your overcloud plan. This includes the plan name, main template to use, and environment files to apply to the overcloud.

capabilities-map.yaml

A mapping of environment files for an overcloud plan. Use this file to describe and enable environment files through the director's web UI. Custom environment files detected in the **environments** directory in an overcloud plan but not defined in the **capabilities-map.yaml** are listed in the **Other** subtab of **2 Specify Deployment Configuration > Overall Settings** on the web UI.

environments

Contains additional Heat environment files that you can use with your Overcloud creation. These environment files enable extra functions for your resulting OpenStack environment. For example, the directory contains an environment file for enabling Cinder NetApp backend storage (**cinder-netapp-config.yaml**). Any environment files detected in this directory that are not defined in the **capabilities-map.yaml** file are listed in the **Other** subtab of **2 Specify Deployment Configuration > Overall Settings** in the director's web UI.

network

A set of Heat templates to help create isolated networks and ports.

puppet

Templates mostly driven by configuration with puppet. The aforementioned **overcloud-resource-registry-puppet.j2.yaml** environment file uses the files in this directory to drive the application of the Puppet configuration on each node.

puppet/services

A directory containing Heat templates for all services in the composable service architecture.

extraconfig

Templates used to enable extra functionality.

firstboot

Provides example **first_boot** scripts that the director uses when initially creating the nodes.

2.4. PLAN ENVIRONMENT METADATA

A plan environment metadata file allows you to define metadata about your overcloud plan. This information is used when importing and exporting your overcloud plan, plus used during the overcloud creation from your plan.

A plan environment metadata file includes the following parameters:

version

The version of the template.

name

The name of the overcloud plan and the container in OpenStack Object Storage (swift) used to store the plan files.

template

The core parent template to use for the overcloud deployment. This is most often **overcloud.yaml**, which is the rendered version of the **overcloud.yaml.j2** template.

environments

Defines a list of environment files to use. Specify the path of each environment file with the **path** sub-parameter.

parameter_defaults

A set of parameters to use in your overcloud. This functions in the same way as the **parameter_defaults** section in a standard environment file.

passwords

A set of parameters to use for overcloud passwords. This functions in the same way as the **parameter_defaults** section in a standard environment file. Usually, the director automatically populates this section with randomly generated passwords.

workflow_parameters

Allows you to provide a set of parameters to OpenStack Workflow (mistral) namespaces. You can use this to calculate and automatically generate certain overcloud parameters.

The following is an example of the syntax of a plan environment file:

```
version: 1.0
name: myovercloud
description: 'My Overcloud Plan'
template: overcloud.yaml
environments:
- path: overcloud-resource-registry-puppet.yaml
- path: environments/containers-default-parameters.yaml
- path: user-environment.yaml
parameter_defaults:
  ControllerCount: 1
  ComputeCount: 1
  OvercloudComputeFlavor: compute
  OvercloudControllerFlavor: control
workflow_parameters:
  tripleo.derive_params.v1.derive_parameters:
    num_phy_cores_per_numa_node_for_pmd: 2
```

You can include the plan environment metadata file with the **openstack overcloud deploy** command using the **-p** option. For example:

```
(undercloud) $ openstack overcloud deploy --templates \
  -p /my-plan-environment.yaml \
  [OTHER OPTIONS]
```

You can also view plan metadata for an existing overcloud plan using the following command:

```
(undercloud) $ openstack object save overcloud plan-environment.yaml --file -
```

2.5. INCLUDING ENVIRONMENT FILES IN OVERCLOUD CREATION

The deployment command (**openstack overcloud deploy**) uses the **-e** option to include an environment file to customize your Overcloud. You can include as many environment files as necessary. However, the order of the environment files is important as the parameters and resources defined in subsequent environment files take precedence. For example, you might have two environment files:

environment-file-1.yaml

```
resource_registry:
  OS::TripleO::NodeExtraConfigPost: /home/stack/templates/template-1.yaml

parameter_defaults:
  RabbitFDLimit: 65536
  TimeZone: 'Japan'
```

environment-file-2.yaml

```
resource_registry:
  OS::TripleO::NodeExtraConfigPost: /home/stack/templates/template-2.yaml
```

```
parameter_defaults:
  TimeZone: 'Hongkong'
```

Then deploy with both environment files included:

```
$ openstack overcloud deploy --templates -e environment-file-1.yaml -e environment-file-2.yaml
```

In this example, both environment files contain a common resource type (**OS::TripleO::NodeExtraConfigPost**) and a common parameter (**TimeZone**). The **openstack overcloud deploy** command runs through the following process:

1. Loads the default configuration from the core Heat template collection as per the **--template** option.
2. Applies the configuration from **environment-file-1.yaml**, which overrides any common settings from the default configuration.
3. Applies the configuration from **environment-file-2.yaml**, which overrides any common settings from the default configuration and **environment-file-1.yaml**.

This results in the following changes to the default configuration of the Overcloud:

- **OS::TripleO::NodeExtraConfigPost** resource is set to **/home/stack/templates/template-2.yaml** as per **environment-file-2.yaml**.
- **TimeZone** parameter is set to **Hongkong** as per **environment-file-2.yaml**.
- **RabbitFDLimit** parameter is set to **65536** as per **environment-file-1.yaml**. **environment-file-2.yaml** does not change this value.

This provides a method for defining custom configuration to the your Overcloud without values from multiple environment files conflicting.

2.6. USING CUSTOMIZED CORE HEAT TEMPLATES

When creating the overcloud, the director uses a core set of Heat templates located in **/usr/share/openstack-tripleo-heat-templates**. If you want to customize this core template collection, use a Git workflow to track changes and merge updates. Use the following git processes to help manage your custom template collection:

Initializing a Custom Template Collection

Use the following procedure to create an initial Git repository containing the Heat template collection:

1. Copy the template collection to the **stack** users directory. This example copies the collection to the **~/templates** directory:

```
$ cd ~/templates
$ cp -r /usr/share/openstack-tripleo-heat-templates .
```

2. Change to the custom template directory and initialize a Git repository:

```
$ cd openstack-tripleo-heat-templates
$ git init .
```

3. Stage all templates for the initial commit:

```
$ git add *
```

4. Create an initial commit:

```
$ git commit -m "Initial creation of custom core heat templates"
```

This creates an initial **master** branch containing the latest core template collection. Use this branch as the basis for your custom branch and merge new template versions to this branch.

Creating a Custom Branch and Committing Changes

Use a custom branch to store your changes to the core template collection. Use the following procedure to create a **my-customizations** branch and add customizations to it:

1. Create the **my-customizations** branch and switch to it:

```
$ git checkout -b my-customizations
```

2. Edit the files in the custom branch.

3. Stage the changes in git:

```
$ git add [edited files]
```

4. Commit the changes to the custom branch:

```
$ git commit -m "[Commit message for custom changes]"
```

This adds your changes as commits to the **my-customizations** branch. When the **master** branch updates, you can rebase **my-customizations** off **master**, which causes git to add these commits on to the updated template collection. This helps track your customizations and replay them on future template updates.

Updating the Custom Template Collection:

When updating the undercloud, the **openstack-tripleo-heat-templates** package might also update. When this occurs, use the following procedure to update your custom template collection:

1. Save the **openstack-tripleo-heat-templates** package version as an environment variable:

```
$ export PACKAGE=$(rpm -qv openstack-tripleo-heat-templates)
```

2. Change to your template collection directory and create a new branch for the updated templates:

```
$ cd ~/templates/openstack-tripleo-heat-templates
$ git checkout -b $PACKAGE
```

3. Remove all files in the branch and replace them with the new versions:

```
$ git rm -rf *
$ cp -r /usr/share/openstack-tripleo-heat-templates/* .
```

4. Add all templates for the initial commit:

```
$ git add *
```

5. Create a commit for the package update:

```
$ git commit -m "Updates for $PACKAGE"
```

6. Merge the branch into master. If you use a Git management system (such as GitLab), use the management workflow. If you use git locally, merge by switching to the **master** branch and run the **git merge** command:

```
$ git checkout master
$ git merge $PACKAGE
```

The **master** branch now contains the latest version of the core template collection. You can now rebase the **my-customization** branch from this updated collection.

Rebasing the Custom Branch

Use the following procedure to update the **my-customization** branch,:

1. Change to the **my-customizations** branch:

```
$ git checkout my-customizations
```

2. Rebase the branch off **master**:

```
$ git rebase master
```

This updates the **my-customizations** branch and replays the custom commits made to this branch.

If git reports any conflicts during the rebase, use this procedure:

1. Check which files contain the conflicts:

```
$ git status
```

2. Resolve the conflicts of the template files identified.

3. Add the resolved files:

```
$ git add [resolved files]
```

4. Continue the rebase:

```
$ git rebase --continue
```

Deploying Custom Templates

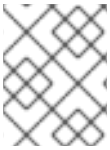
Use the following procedure to deploy the custom template collection:

1. Ensure that you have switched to the **my-customization** branch:

```
git checkout my-customizations
```

2. Run the **openstack overcloud deploy** command with the **--templates** option to specify your local template directory:

```
$ openstack overcloud deploy --templates /home/stack/templates/openstack-tripleo-heat-templates [OTHER OPTIONS]
```



NOTE

The director uses the default template directory (**/usr/share/openstack-tripleo-heat-templates**) if you specify the **--templates** option without a directory.

IMPORTANT

Red Hat recommends using the methods in Chapter 4, *Configuration Hooks* instead of modifying the Heat template collection.

2.7. JINJA2 RENDERING

The core Heat templates in **/usr/share/openstack-tripleo-heat-templates** contains a number of files ending with a **j2.yaml** extension. These files contain Jinja2 template syntax and the director renders these files to their static Heat template equivalents ending in **.yaml**. For example, the main **overcloud.j2.yaml** file renders into **overcloud.yaml**. The director uses the resulting **overcloud.yaml** file.

The Jinja2-enabled Heat templates use Jinja2 syntax to create parameters and resources for iterative values. For example, the **overcloud.j2.yaml** file contains the following snippet:

```
parameters:
...
{% for role in roles %}
  ...
  {{role.name}}Count:
    description: Number of {{role.name}} nodes to deploy
    type: number
    default: {{role.CountDefault|default(0)}}
  ...
{% endfor %}
```

When the director renders the Jinja2 syntax, the director iterates over the roles defined in the **roles_data.yaml** file and populates the **{{role.name}}Count** parameter with the name of the role. The default **roles_data.yaml** file contains five roles and results in the the following parameters from our example:

- **ControllerCount**
- **ComputeCount**
- **BlockStorageCount**
- **ObjectStorageCount**
- **CephStorageCount**

A example rendered version of the parameter looks like this:

```
parameters:
  ...
  ControllerCount:
    description: Number of Controller nodes to deploy
    type: number
    default: 1
  ...
```

The director only renders Jinja2-enabled templates and environment files within the directory of your core Heat templates. The following use cases demonstrate the correct method to render the Jinja2 templates.

Use case 1: Default core templates

Template directory: /usr/share/openstack-tripleo-heat-templates/

Environment file: /usr/share/openstack-tripleo-heat-templates/environments/network-isolation.j2.yaml

The director uses the default core template location (**--templates**). The director renders the **network-isolation.j2.yaml** file into **network-isolation.yaml**. When running the **openstack overcloud deploy** command, use the **-e** option to include the name of rendered **network-isolation.yaml** file.

```
$ openstack ovecloud deploy --templates \
    -e /usr/share/openstack-tripleo-heat-templates/environments/network-isolation.yaml
    ...
```

Use case 2: Custom core templates

Template directory: /home/stack/tripleo-heat-templates

Environment file: /home/stack/tripleo-heat-templates/environments/network-isolation.j2.yaml

The director uses a custom core template location (**--templates /home/stack/tripleo-heat-templates**). The director renders the **network-isolation.j2.yaml** file within the custom core templates into **network-isolation.yaml**. When running the **openstack overcloud deploy** command, use the **-e** option to include the name of rendered **network-isolation.yaml** file.

```
$ openstack ovecloud deploy --templates /home/stack/tripleo-heat-templates \
    -e /home/stack/tripleo-heat-templates/environments/network-isolation.yaml
    ...
```

Use case 3: Incorrect usage

Template directory: /usr/share/openstack-tripleo-heat-templates/

Environment file: /home/stack/tripleo-heat-templates/environments/network-isolation.j2.yaml

This director uses a custom core template location (**--templates /home/stack/tripleo-heat-templates**). However, the chosen **network-isolation.j2.yaml** is not located within the custom core templates, so it *will not* render into **network-isolation.yaml**. This causes the deployment to fail.

CHAPTER 3. PARAMETERS

Each Heat template in the director's template collection contains a **parameters** section. This section defines all parameters specific to a particular overcloud service. This includes the following:

- **overcloud.j2.yaml** - Default base parameters
- **roles_data.yaml** - Default parameters for composable roles
- **deployment/*.yaml** - Default parameters for specific services

You can modify the values for these parameters using the following method:

1. Create an environment file for your custom parameters.
2. Include your custom parameters in the **parameter_defaults** section of the environment file.
3. Include the environment file with the **openstack overcloud deploy** command.

The next few sections contain examples to demonstrate how to configure specific parameters for services in the **deployment** directory.

3.1. EXAMPLE 1: CONFIGURING THE TIME ZONE

The Heat template for setting the time zone (**deployment/time/timezone-baremetal-ansible.yaml**) contains a **TimeZone** parameter. If you leave the **TimeZone** parameter blank, the overcloud sets the time zone to **UTC** by default. The director recognizes the standard time zone names defined in the time zone database **/usr/share/zoneinfo/**. For example, if you want to set your time zone to **Japan**, examine the contents of **/usr/share/zoneinfo** to locate a suitable entry:

```
$ ls /usr/share/zoneinfo/
Africa      Asia      Canada    Cuba      EST       GB        GMT-0     HST       iso3166.tab  Kwajalein  MST
NZ-CHAT     posix     right     Turkey    UTC       Zulu
America     Atlantic  CET       EET       EST5EDT   GB-Eire   GMT+0     Iceland   Israel       Libya
MST7MDT     Pacific   posixrules ROC      UCT       WET
Antarctica  Australia Chile     Egypt     Etc       GMT       Greenwich Indian    Jamaica      MET        Navajo
Poland      PRC       ROK       Universal W-SU
Arctic      Brazil    CST6CDT   Eire      Europe    GMT0      Hongkong  Iran      Japan        Mexico     NZ
Portugal    PST8PDT   Singapore US        zone.tab
```

The output listed above includes time zone files and directories containing additional time zone files. For example, **Japan** is an individual time zone file in this result, but **Africa** is a directory containing additional time zone files:

```
$ ls /usr/share/zoneinfo/Africa/
Abidjan     Algiers   Bamako    Bissau    Bujumbura  Ceuta     Dar_es_Salaam  El_Aaiun  Harare
Kampala     Kinshasa  Lome      Lusaka    Maseru     Monrovia  Niamey         Porto-Novo  Tripoli
Accra       Asmara    Bangui    Blantyre  Cairo      Conakry   Djibouti       Freetown  Johannesburg
Khartoum    Lagos     Luanda    Malabo    Mbabane    Nairobi   Nouakchott     Sao_Tome  Tunis
Addis_Ababa Asmera    Banjul    Brazzaville Casablanca Dakar   Douala         Gaborone  Juba
Kigali      Libreville Lubumbashi Maputo  Mogadishu  Ndjamena  Ouagadougou    Timbuktu
Windhoek
```

Add the entry in an environment file to set your time zone to **Japan**:

```
parameter_defaults:
  TimeZone: 'Japan'
```

3.2. EXAMPLE 2: ENABLING NETWORKING DISTRIBUTED VIRTUAL ROUTING (DVR)

The Heat template for the OpenStack Networking (neutron) API (**deployment/neutron/neutron-api-container-puppet.yaml**) contains a parameter to enable and disable Distributed Virtual Routing (DVR). The default for the parameter is **false**. To enable it, use the following in an environment file:

```
parameter_defaults:
  NeutronEnableDVR: true
```

3.3. EXAMPLE 3: CONFIGURING RABBITMQ FILE DESCRIPTOR LIMIT

For certain configurations, you might need to increase the file descriptor limit for the RabbitMQ server. The **deployment/rabbitmq/rabbitmq-container-puppet.yaml** Heat template allows you to set the **RabbitFDLimit** parameter to the limit you require. Add the following to an environment file:

```
parameter_defaults:
  RabbitFDLimit: 65536
```

3.4. EXAMPLE 4: ENABLING AND DISABLING PARAMETERS

You might need to initially set a parameter during a deployment, then disable the parameter for a future deployment operation, such as updates or scaling operations. For example, to include a custom RPM during the overcloud creation, include the following:

```
parameter_defaults:
  DeployArtifactURLs: ["http://www.example.com/myfile.rpm"]
```

To disable this parameter from a future deployment, it is not enough to remove the parameter. Instead, you set the parameter to an empty value:

```
parameter_defaults:
  DeployArtifactURLs: []
```

This ensures the parameter is no longer set for subsequent deployments operations.

3.5. IDENTIFYING PARAMETERS TO MODIFY

Red Hat OpenStack Platform director provides many parameters for configuration. In some cases, you might experience difficulty identifying a certain option to configure and the corresponding director parameter. If there is an option you want to configure through the director, use the following workflow to identify and map the option to a specific overcloud parameter:

1. Identify the option you aim to configure. Make a note of the service that uses the option.
2. Check the corresponding Puppet module for this option. The Puppet modules for Red Hat OpenStack Platform are located under **/etc/puppet/modules** on the director node. Each module corresponds to a particular service. For example, the **keystone** module corresponds to

the OpenStack Identity (keystone).

- If the Puppet module contains a variable that controls the chosen option, move to the next step.
- If the Puppet module does not contain a variable that controls the chosen option, no hieradata exists for this option. If possible, you can set the option manually after the overcloud completes deployment.

3. Check the director's core Heat template collection for the Puppet variable in the form of hieradata. The templates in **deployment/*** usually correspond to the Puppet modules of the same services. For example, the **deployment/keystone/keystone-container-puppet.yaml** template provides hieradata to the **keystone** module.
   - If the Heat template sets hieradata for the Puppet variable, the template should also disclose the director-based parameter to modify.
   - If the Heat template does not set hieradata for the Puppet variable, use the configuration hooks to pass the hieradata using an environment file. See Section 4.5, "Puppet: Customizing Hieradata for Roles" for more information on customizing hieradata.

Workflow Example

To change the notification format for OpenStack Identity (keystone), use the workflow and complete the following steps:

1. Identify the OpenStack parameter to configure (**notification_format**).
2. Search the **keystone** Puppet module for the **notification_format** setting. For example:

   ```
   $ grep notification_format /etc/puppet/modules/keystone/manifests/*
   ```

   In this case, the **keystone** module manages this option using the **keystone::notification_format** variable.
3. Search the **keystone** service template for this variable. For example:

   ```
   $ grep "keystone::notification_format" /usr/share/openstack-tripleo-heat-templates/deployment/keystone/keystone-container-puppet.yaml
   ```

   The output shows the director using the **KeystoneNotificationFormat** parameter to set the **keystone::notification_format** hieradata.

The following table shows the eventual mapping:

Director Parameter	Puppet Hieradata	OpenStack Identity (keystone) option
KeystoneNotificationFormat	**keystone::notification_format**	**notification_format**

You set the **KeystoneNotificationFormat** in an overcloud's environment file which in turn sets the **notification_format** option in the **keystone.conf** file during the overcloud's configuration.

CHAPTER 4. CONFIGURATION HOOKS

The configuration hooks provide a method to inject your own configuration functions into the Overcloud deployment process. This includes hooks for injecting custom configuration before and after the main Overcloud services configuration and hook for modifying and including Puppet-based configuration.

4.1. FIRST BOOT: CUSTOMIZING FIRST BOOT CONFIGURATION

The director provides a mechanism to perform configuration on all nodes upon the initial creation of the Overcloud. The director achieves this through **cloud-init**, which you can call using the **OS::TripleO::NodeUserData** resource type.

In this example, update the nameserver with a custom IP address on all nodes. First, create a basic Heat template (**/home/stack/templates/nameserver.yaml**) that runs a script to append each node's **resolv.conf** with a specific nameserver. You can use the **OS::TripleO::MultipartMime** resource type to send the configuration script.

```
heat_template_version: 2014-10-16

description: >
  Extra hostname configuration

resources:
  userdata:
    type: OS::Heat::MultipartMime
    properties:
      parts:
      - config: {get_resource: nameserver_config}

  nameserver_config:
    type: OS::Heat::SoftwareConfig
    properties:
      config: |
        #!/bin/bash
        echo "nameserver 192.168.1.1" >> /etc/resolv.conf

outputs:
  OS::stack_id:
    value: {get_resource: userdata}
```

Create an environment file (**/home/stack/templates/firstboot.yaml**) that registers your Heat template as the **OS::TripleO::NodeUserData** resource type.

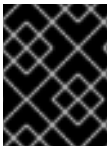
```
resource_registry:
  OS::TripleO::NodeUserData: /home/stack/templates/nameserver.yaml
```

To add the first boot configuration, add the environment file to the stack along with your other environment files when first creating the Overcloud. For example:

```
$ openstack overcloud deploy --templates \
    ...
    -e /home/stack/templates/firstboot.yaml \
    ...
```

The **-e** applies the environment file to the Overcloud stack.

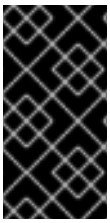
This adds the configuration to all nodes when they are first created and boot for the first time. Subsequent inclusions of these templates, such as updating the Overcloud stack, does not run these scripts.



IMPORTANT

You can only register the **OS::TripleO::NodeUserData** to one Heat template. Subsequent usage overrides the Heat template to use.

4.2. PRE-CONFIGURATION: CUSTOMIZING SPECIFIC OVERCLOUD ROLES



IMPORTANT

Previous versions of this document used the **OS::TripleO::Tasks::*PreConfig** resources to provide pre-configuration hooks on a per role basis. The director's Heat template collection requires dedicated use of these hooks, which means you should not use them for custom use. Instead, use the **OS::TripleO::*ExtraConfigPre** hooks outlined below.

The Overcloud uses Puppet for the core configuration of OpenStack components. The director provides a set of hooks to provide custom configuration for specific node roles after the first boot completes and before the core configuration begins. These hooks include:

OS::TripleO::ControllerExtraConfigPre

Additional configuration applied to Controller nodes before the core Puppet configuration.

OS::TripleO::ComputeExtraConfigPre

Additional configuration applied to Compute nodes before the core Puppet configuration.

OS::TripleO::CephStorageExtraConfigPre

Additional configuration applied to Ceph Storage nodes before the core Puppet configuration.

OS::TripleO::ObjectStorageExtraConfigPre

Additional configuration applied to Object Storage nodes before the core Puppet configuration.

OS::TripleO::BlockStorageExtraConfigPre

Additional configuration applied to Block Storage nodes before the core Puppet configuration.

OS::TripleO::[ROLE]ExtraConfigPre

Additional configuration applied to custom nodes before the core Puppet configuration. Replace **[ROLE]** with the composable role name.

In this example, you first create a basic Heat template (**/home/stack/templates/nameserver.yaml**) that runs a script to write to a node's **resolv.conf** with a variable nameserver.

```
heat_template_version: 2014-10-16

description: >
  Extra hostname configuration

  parameters:
    server:
      type: json
```

```
    nameserver_ip:
      type: string
    DeployIdentifier:
      type: string

resources:
  CustomExtraConfigPre:
    type: OS::Heat::SoftwareConfig
    properties:
      group: script
      config:
        str_replace:
          template: |
            #!/bin/sh
            echo "nameserver _NAMESERVER_IP_" > /etc/resolv.conf
          params:
            _NAMESERVER_IP_: {get_param: nameserver_ip}

  CustomExtraDeploymentPre:
    type: OS::Heat::SoftwareDeployment
    properties:
      server: {get_param: server}
      config: {get_resource: CustomExtraConfigPre}
      actions: ['CREATE','UPDATE']
      input_values:
        deploy_identifier: {get_param: DeployIdentifier}

outputs:
  deploy_stdout:
    description: Deployment reference, used to trigger pre-deploy on changes
    value: {get_attr: [CustomExtraDeploymentPre, deploy_stdout]}
```

In this example, the **resources** section contains the following parameters:

CustomExtraConfigPre

This defines a software configuration. In this example, we define a Bash **script** and Heat replaces **_NAMESERVER_IP_** with the value stored in the **nameserver_ip** parameter.

CustomExtraDeploymentPre

This executes a software configuration, which is the software configuration from the **CustomExtraConfigPre** resource. Note the following:

- The **config** parameter makes a reference to the **CustomExtraConfigPre** resource so Heat knows what configuration to apply.
- The **server** parameter retrieves a map of the Overcloud nodes. This parameter is provided by the parent template and is mandatory in templates for this hook.
- The **actions** parameter defines when to apply the configuration. In this case, apply the configuration when the Overcloud is created. Possible actions include **CREATE**, **UPDATE**, **DELETE**, **SUSPEND**, and **RESUME**.
- **input_values** contains a parameter called **deploy_identifier**, which stores the **DeployIdentifier** from the parent template. This parameter provides a timestamp to the resource for each deployment update. This ensures the resource reapplies on subsequent overcloud updates.

Create an environment file (/**home**/**stack**/**templates**/**pre_config.yaml**) that registers your Heat template to the role-based resource type. For example, to apply only to Controller nodes, use the **ControllerExtraConfigPre** hook:

```
resource_registry:
  OS::TripleO::ControllerExtraConfigPre: /home/stack/templates/nameserver.yaml

parameter_defaults:
  nameserver_ip: 192.168.1.1
```

To apply the configuration, add the environment file to the stack along with your other environment files when creating or updating the Overcloud. For example:

```
$ openstack overcloud deploy --templates \
    ...
    -e /home/stack/templates/pre_config.yaml \
    ...
```

This applies the configuration to all Controller nodes before the core configuration begins on either the initial Overcloud creation or subsequent updates.

IMPORTANT

You can only register each resource to only one Heat template per hook. Subsequent usage overrides the Heat template to use.

4.3. PRE-CONFIGURATION: CUSTOMIZING ALL OVERCLOUD ROLES

The Overcloud uses Puppet for the core configuration of OpenStack components. The director provides a hook to configure all node types after the first boot completes and before the core configuration begins:

OS::TripleO::NodeExtraConfig

Additional configuration applied to all nodes roles before the core Puppet configuration.

In this example, create a basic Heat template (/**home**/**stack**/**templates**/**nameserver.yaml**) that runs a script to append each node's **resolv.conf** with a variable nameserver.

```
heat_template_version: 2014-10-16

description: >
  Extra hostname configuration

parameters:
  server:
    type: string
  nameserver_ip:
    type: string
  DeployIdentifier:
    type: string

resources:
  CustomExtraConfigPre:
    type: OS::Heat::SoftwareConfig
```

```
    properties:
      group: script
      config:
        str_replace:
          template: |
            #!/bin/sh
            echo "nameserver _NAMESERVER_IP_" >> /etc/resolv.conf
          params:
            _NAMESERVER_IP_: {get_param: nameserver_ip}

  CustomExtraDeploymentPre:
    type: OS::Heat::SoftwareDeployment
    properties:
      server: {get_param: server}
      config: {get_resource: CustomExtraConfigPre}
      actions: ['CREATE','UPDATE']
      input_values:
        deploy_identifier: {get_param: DeployIdentifier}

outputs:
  deploy_stdout:
    description: Deployment reference, used to trigger pre-deploy on changes
    value: {get_attr: [CustomExtraDeploymentPre, deploy_stdout]}
```

In this example, the **resources** section contains the following parameters:

CustomExtraConfigPre

This defines a software configuration. In this example, we define a Bash **script** and Heat replaces **_NAMESERVER_IP_** with the value stored in the **nameserver_ip** parameter.

CustomExtraDeploymentPre

This executes a software configuration, which is the software configuration from the **CustomExtraConfigPre** resource. Note the following:

- The **config** parameter makes a reference to the **CustomExtraConfigPre** resource so Heat knows what configuration to apply.
- The **server** parameter retrieves a map of the Overcloud nodes. This parameter is provided by the parent template and is mandatory in templates for this hook.
- The **actions** parameter defines when to apply the configuration. In this case, we only apply the configuration when the Overcloud is created. Possible actions include **CREATE**, **UPDATE**, **DELETE**, **SUSPEND**, and **RESUME**.
- The **input_values** parameter contains a sub-parameter called **deploy_identifier**, which stores the **DeployIdentifier** from the parent template. This parameter provides a timestamp to the resource for each deployment update. This ensures the resource reapplies on subsequent overcloud updates.

Next, create an environment file (**/home/stack/templates/pre_config.yaml**) that registers your heat template as the **OS::TripleO::NodeExtraConfig** resource type.

```
resource_registry:
  OS::TripleO::NodeExtraConfig: /home/stack/templates/nameserver.yaml
```

```
parameter_defaults:
  nameserver_ip: 192.168.1.1
```

To apply the configuration, add the environment file to the stack along with your other environment files when creating or updating the Overcloud. For example:

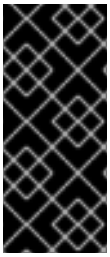
```
$ openstack overcloud deploy --templates \
    ...
    -e /home/stack/templates/pre_config.yaml \
    ...
```

This applies the configuration to all nodes before the core configuration begins on either the initial Overcloud creation or subsequent updates.

IMPORTANT

You can only register the **OS::TripleO::NodeExtraConfig** to only one Heat template. Subsequent usage overrides the Heat template to use.

4.4. POST-CONFIGURATION: CUSTOMIZING ALL OVERCLOUD ROLES



IMPORTANT

Previous versions of this document used the **OS::TripleO::Tasks::*PostConfig** resources to provide post-configuration hooks on a per role basis. The director's Heat template collection requires dedicated use of these hooks, which means you should not use them for custom use. Instead, use the **OS::TripleO::NodeExtraConfigPost** hook outlined below.

A situation might occur where you have completed the creation of your Overcloud but want to add additional configuration to all roles, either on initial creation or on a subsequent update of the Overcloud. In this case, you use the following post-configuration hook:

OS::TripleO::NodeExtraConfigPost

Additional configuration applied to all nodes roles after the core Puppet configuration.

In this example, you first create a basic heat template (**/home/stack/templates/nameserver.yaml**) that runs a script to append each node's **resolv.conf** with a variable nameserver.

```
heat_template_version: 2014-10-16

description: >
  Extra hostname configuration

parameters:
  servers:
    type: json
  nameserver_ip:
    type: string
  DeployIdentifier:
    type: string
  EndpointMap:
    default: {}
```

```
    type: json

resources:
  CustomExtraConfig:
    type: OS::Heat::SoftwareConfig
    properties:
      group: script
      config:
        str_replace:
          template: |
            #!/bin/sh
            echo "nameserver _NAMESERVER_IP_" >> /etc/resolv.conf
          params:
            _NAMESERVER_IP_: {get_param: nameserver_ip}

  CustomExtraDeployments:
    type: OS::Heat::SoftwareDeploymentGroup
    properties:
      servers:  {get_param: servers}
      config: {get_resource: CustomExtraConfig}
      actions: ['CREATE','UPDATE']
      input_values:
        deploy_identifier: {get_param: DeployIdentifier}
```

In this example, the **resources** section contains the following:

CustomExtraConfig

This defines a software configuration. In this example, we define a Bash **script** and Heat replaces **_NAMESERVER_IP_** with the value stored in the **nameserver_ip** parameter.

CustomExtraDeployments

This executes a software configuration, which is the software configuration from the **CustomExtraConfig** resource. Note the following:

- The **config** parameter makes a reference to the **CustomExtraConfig** resource so Heat knows what configuration to apply.
- The **servers** parameter retrieves a map of the Overcloud nodes. This parameter is provided by the parent template and is mandatory in templates for this hook.
- The **actions** parameter defines when to apply the configuration. In this case, we apply the configuration when the Overcloud is created. Possible actions include **CREATE**, **UPDATE**, **DELETE**, **SUSPEND**, and **RESUME**.
- **input_values** contains a parameter called **deploy_identifier**, which stores the **DeployIdentifier** from the parent template. This parameter provides a timestamp to the resource for each deployment update. This ensures the resource reapplies on subsequent overcloud updates.

Create an environment file (**/home/stack/templates/post_config.yaml**) that registers your Heat template as the **OS::TripleO::NodeExtraConfigPost:** resource type.

```
resource_registry:
  OS::TripleO::NodeExtraConfigPost: /home/stack/templates/nameserver.yaml
```

```
parameter_defaults:
  nameserver_ip: 192.168.1.1
```

To apply the configuration, add the environment file to the stack along with your other environment files when creating or updating the Overcloud. For example:

```
$ openstack overcloud deploy --templates \
    ...
    -e /home/stack/templates/post_config.yaml \
    ...
```

This applies the configuration to all nodes after the core configuration completes on either initial Overcloud creation or subsequent updates.

IMPORTANT

You can only register the **OS::TripleO::NodeExtraConfigPost** to only one Heat template. Subsequent usage overrides the Heat template to use.

4.5. PUPPET: CUSTOMIZING HIERADATA FOR ROLES

The Heat template collection contains a set of parameters to pass extra configuration to certain node types. These parameters save the configuration as hieradata for the node's Puppet configuration. These parameters are:

ControllerExtraConfig

Configuration to add to all Controller nodes.

ComputeExtraConfig

Configuration to add to all Compute nodes.

BlockStorageExtraConfig

Configuration to add to all Block Storage nodes.

ObjectStorageExtraConfig

Configuration to add to all Object Storage nodes.

CephStorageExtraConfig

Configuration to add to all Ceph Storage nodes.

[ROLE]ExtraConfig

Configuration to add to a composable role. Replace **[ROLE]** with the composable role name.

ExtraConfig

Configuration to add to all nodes.

To add extra configuration to the post-deployment configuration process, create an environment file that contains these parameters in the **parameter_defaults** section. For example, to increase the reserved memory for Compute hosts to 1024 MB and set the VNC keymap to Japanese:

```
parameter_defaults:
  ComputeExtraConfig:
    nova::compute::reserved_host_memory: 1024
    nova::compute::vnc_keymap: ja
```

Include this environment file when running **openstack overcloud deploy**.

IMPORTANT

You can only define each parameter once. Subsequent usage overrides previous values.

4.6. PUPPET: CUSTOMIZING HIERADATA FOR INDIVIDUAL NODES

You can set Puppet hieradata for individual nodes using the Heat template collection. To accomplish this, acquire the system UUID saved as part of the introspection data for a node:

```
$ openstack baremetal introspection data save 9dcc87ae-4c6d-4ede-81a5-9b20d7dc4a14 | jq
.extra.system.product.uuid
```

This outputs a system UUID. For example:

```
"F5055C6C-477F-47FB-AFE5-95C6928C407F"
```

Use this system UUID in an environment file that defines node-specific hieradata and registers the **per_node.yaml** template to a pre-configuration hook. For example:

```
resource_registry:
  OS::TripleO::ComputeExtraConfigPre: /usr/share/openstack-tripleo-heat-
templates/puppet/extraconfig/pre_deploy/per_node.yaml
parameter_defaults:
  NodeDataLookup: '{"F5055C6C-477F-47FB-AFE5-95C6928C407F":
{"nova::compute::vcpu_pin_set": [ "2", "3" ]}}'
```

Include this environment file when running **openstack overcloud deploy**.

The **per_node.yaml** template generates a set of heiradata files on nodes that correspond to each system UUID and contains the hieradata you defined. If a UUID is not defined, the resulting hieradata file is empty. In the previous example, the **per_node.yaml** template runs on all Compute nodes (as per the **OS::TripleO::ComputeExtraConfigPre** hook), but only the Compute node with system UUID **F5055C6C-477F-47FB-AFE5-95C6928C407F** receives hieradata.

This provides a method of tailoring each node to specific requirements.

For more information about NodeDataLookup, see section Mapping the Disk Layout to Non-Homogeneous Ceph Storage Nodes of the *Storage Guide*.

4.7. PUPPET: APPLYING CUSTOM MANIFESTS

In certain circumstances, you might need to install and configure some additional components to your Overcloud nodes. You can achieve this with a custom Puppet manifest that applies to nodes after the main configuration completes. As a basic example, you might intend to install **motd** to each node. The process for accomplishing this is to first create a Heat template (**/home/stack/templates/custom_puppet_config.yaml**) that launches Puppet configuration.

```
heat_template_version: 2014-10-16

description: >
  Run Puppet extra configuration to set new MOTD

parameters:
  servers:
```

```
    type: json

resources:
  ExtraPuppetConfig:
    type: OS::Heat::SoftwareConfig
    properties:
      config: {get_file: motd.pp}
      group: puppet
      options:
        enable_hiera: True
        enable_facter: False

  ExtraPuppetDeployments:
    type: OS::Heat::SoftwareDeploymentGroup
    properties:
      config: {get_resource: ExtraPuppetConfig}
      servers: {get_param: servers}
```

This includes the **/home/stack/templates/motd.pp** within the template and passes it to nodes for configuration. The **motd.pp** file itself contains the Puppet classes to install and configure **motd**.

Create an environment file (**/home/stack/templates/puppet_post_config.yaml**) that registers your heat template as the **OS::TripleO::NodeExtraConfigPost:** resource type.

```
resource_registry:
  OS::TripleO::NodeExtraConfigPost: /home/stack/templates/custom_puppet_config.yaml
```

Include this environment file along with your other environment files when creating or updating the Overcloud stack:

```
$ openstack overcloud deploy --templates \
    ...
    -e /home/stack/templates/puppet_post_config.yaml \
    ...
```

This applies the configuration from **motd.pp** to all nodes in the Overcloud.

CHAPTER 5. ANSIBLE-BASED OVERCLOUD REGISTRATION

The director uses Ansible-based methods to register overcloud nodes to the Red Hat Customer Portal or a Red Hat Satellite 6 server.

5.1. RHSM COMPOSABLE SERVICE

The **rhsm** composable service provides a method to register overcloud nodes through Ansible. Each role in the default **roles_data** file contains a **OS::TripleO::Services::Rhsm** resource, which is disabled by default. To enable the service, register the resource to the **rhsm** composable service file:

```
resource_registry:
  OS::TripleO::Services::Rhsm: /usr/share/openstack-tripleo-heat-
templates/extraconfig/services/rhsm.yaml
```

The **rhsm** composable service accepts a **RhsmVars** parameter, which allows you to define multiple sub-parameters relevant to your registration. For example:

```
parameter_defaults:
  RhsmVars:
    rhsm_repos:
      - rhel-8-for-x86_64-baseos-rpms
      - rhel-8-for-x86_64-appstream-rpms
      - rhel-8-for-x86_64-highavailability-rpms
      - ansible-2.8-for-rhel-8-x86_64-rpms
      - openstack-beta-for-rhel-8-x86_64-rpms
      - rhceph-4-osd-for-rhel-8-x86_64-rpms
      - rhceph-4-mon-for-rhel-8-x86_64-rpms
      - rhceph-4-tools-for-rhel-8-x86_64-rpms
    rhsm_username: "myusername"
    rhsm_password: "p@55w0rd!"
    rhsm_org_id: "1234567"
```

You can also use the **RhsmVars** parameter in combination with role-specific parameters (e.g. **ControllerParameters**) to provide flexibility when enabling specific repositories for different nodes types.

The next section is a list of sub-parameters available to use with the **RhsmVars** parameter for use with the **rhsm** composable service.

5.2. RHSMVARS SUB-PARAMETERS

See the role documentation to learn about all Ansible parameters.

rhsm	Description
rhsm_method	Choose the registration method. Either **portal**, **satellite**, or **disable**.
rhsm_org_id	The organization to use for registration. To locate this ID, run **sudo subscription-manager orgs** from the undercloud node. Enter your Red Hat credentials when prompted, and use the resulting **Key** value.

rhsm	Description
rhsm_pool_ids	The subscription pool ID to use. Use this if not auto-attaching subscriptions. To locate this ID, run **sudo subscription-manager list --available --all --matches="*OpenStack*"** from the undercloud node, and use the resulting **Pool ID** value.
rhsm_activation_key	The activation key to use for registration. Does not work when **rhsm_repos** is configured.
rhsm_autosubscribe	Automatically attach compatible subscriptions to this system. Set to **true** to enable.
rhsm_satellite_url	The base URL of the Satellite server to register Overcloud nodes.
rhsm_repos	A list of repositories to enable. Does not work when **rhsm_activation_key** is configured.
rhsm_username	The username for registration. If possible, use activation keys for registration.
rhsm_password	The password for registration. If possible, use activation keys for registration.
rhsm_rhsm_proxy_hostname	The hostname for the HTTP proxy. For example: **proxy.example.com**.
rhsm_rhsm_proxy_port	The port for HTTP proxy communication. For example: **8080**.
rhsm_rhsm_proxy_user	The username to access the HTTP proxy.
rhsm_rhsm_proxy_password	The password to access the HTTP proxy.

Now that you have an understanding of how the **rhsm** composable service works and how to configure it, you can use the following procedures to configure your own registration details.

5.3. REGISTERING THE OVERCLOUD WITH THE RHSM COMPOSABLE SERVICE

Use the following procedure to create an environment file that enables and configures the **rhsm** composable service. The director uses this environment file to register and subscribe your nodes.

Procedure

1. Create an environment file (**templates/rhsm.yml**) to store the configuration.
2. Include your configuration in the environment file. For example:

```
resource_registry:
```

```
  OS::TripleO::Services::Rhsm: /usr/share/openstack-tripleo-heat-
templates/extraconfig/services/rhsm.yaml
parameter_defaults:
  RhsmVars:
    rhsm_repos:
      - rhel-8-for-x86_64-baseos-rpms
      - rhel-8-for-x86_64-appstream-rpms
      - rhel-8-for-x86_64-highavailability-rpms
      - ansible-2.8-for-rhel-8-x86_64-rpms
      - openstack-beta-for-rhel-8-x86_64-rpms
      - rhceph-4-osd-for-rhel-8-x86_64-rpms
      - rhceph-4-mon-for-rhel-8-x86_64-rpms
      - rhceph-4-tools-for-rhel-8-x86_64-rpms
    rhsm_username: "myusername"
    rhsm_password: "p@55w0rd!"
    rhsm_org_id: "1234567"
    rhsm_pool_ids: "1a85f9223e3d5e43013e3d6e8ff506fd"
    rhsm_method: "portal"
```

The **resource_registry** associates the **rhsm** composable service with the **OS::TripleO::Services::Rhsm** resource, which is available on each role.

The **RhsmVars** variable passes parameters to Ansible for configuring your Red Hat registration.

3. Save the environment file.

You can also provide registration details to specific overcloud roles. The next section provides an example of this.

5.4. APPLYING THE RHSM COMPOSABLE SERVICE TO DIFFERENT ROLES

You can apply the **rhsm** composable service on a per-role basis. For example, you can apply one set of configuration to Controller nodes and a different set of configuration to Compute nodes.

Procedure

1. Create an environment file (**templates/rhsm.yml**) to store the configuration.

2. Include your configuration in the environment file. For example:

```
resource_registry:
  OS::TripleO::Services::Rhsm: /usr/share/openstack-tripleo-heat-
templates/extraconfig/services/rhsm.yaml
parameter_defaults:
  ControllerParameters:
    RhsmVars:
      rhsm_repos:
        - rhel-8-for-x86_64-baseos-rpms
        - rhel-8-for-x86_64-appstream-rpms
        - rhel-8-for-x86_64-highavailability-rpms
        - ansible-2.8-for-rhel-8-x86_64-rpms
        - openstack-beta-for-rhel-8-x86_64-rpms
        - rhceph-4-osd-for-rhel-8-x86_64-rpms
        - rhceph-4-mon-for-rhel-8-x86_64-rpms
```

```
      - rhceph-4-tools-for-rhel-8-x86_64-rpms
    rhsm_username: "myusername"
    rhsm_password: "p@55w0rd!"
    rhsm_org_id: "1234567"
    rhsm_pool_ids: "1a85f9223e3d5e43013e3d6e8ff506fd"
    rhsm_method: "portal"
ComputeParameters:
  RhsmVars:
    rhsm_repos:
      - rhel-8-for-x86_64-baseos-rpms
      - rhel-8-for-x86_64-appstream-rpms
      - rhel-8-for-x86_64-highavailability-rpms
      - ansible-2.8-for-rhel-8-x86_64-rpms
      - openstack-beta-for-rhel-8-x86_64-rpms
      - rhceph-4-tools-for-rhel-8-x86_64-rpms
    rhsm_username: "myusername"
    rhsm_password: "p@55w0rd!"
    rhsm_org_id: "1234567"
    rhsm_pool_ids: "1a85f9223e3d5e43013e3d6e8ff506fd"
    rhsm_method: "portal"
```

The **resource_registry** associates the **rhsm** composable service with the **OS::TripleO::Services::Rhsm** resource, which is available on each role.

Both **ControllerParameters** and **ComputeParameters** use their own **RhsmVars** parameter to pass subscription details to their respective roles.

3. Save the environment file.

These procedures enable and configure **rhsm** on the overcloud. However, if you used the **rhel-registration** method from previous Red Hat OpenStack Platform version, you must disable it and switch to the Ansible-based method. Use the following procedure to switch from the old **rhel-registration** method to the Ansible-based method.

5.5. SWITCHING TO THE RHSM COMPOSABLE SERVICE

The previous **rhel-registration** method runs a bash script to handle the overcloud registration. The scripts and environment files for this method are located in the core Heat template collection at **/usr/share/openstack-tripleo-heat-templates/extraconfig/pre_deploy/rhel-registration/**.

Complete the following steps to switch from the **rhel-registration** method to the **rhsm** composable service.

Procedure

1. Exclude the **rhel-registration** environment files from future deployments operations. In most cases, exclude the following files:
   - **rhel-registration/environment-rhel-registration.yaml**
   - **rhel-registration/rhel-registration-resource-registry.yaml**
2. If you use a custom **roles_data** file, ensure that each role in your **roles_data** file contains the **OS::TripleO::Services::Rhsm** composable service. For example:

```
- name: Controller
```

```
    description: |
      Controller role that has all the controller services loaded and handles
      Database, Messaging and Network functions.
    CountDefault: 1
    ...
    ServicesDefault:
      ...
      - OS::TripleO::Services::Rhsm
      ...
```

3. Add the environment file for **rhsm** composable service parameters to future deployment operations.

This method replaces the **rhel-registration** parameters with the **rhsm** service parameters and changes the Heat resource that enables the service from:

```
resource_registry:
  OS::TripleO::NodeExtraConfig: rhel-registration.yaml
```

To:

```
resource_registry:
  OS::TripleO::Services::Rhsm: /usr/share/openstack-tripleo-heat-
templates/extraconfig/services/rhsm.yaml
```

You can also include the **/usr/share/openstack-tripleo-heat-templates/environments/rhsm.yaml** environment file with your deployment to enable the service.

To help transition your details from the **rhel-registration** method to the **rhsm** method, use the following table to map the your parameters and their values.

5.6. RHEL-REGISTRATION TO RHSM MAPPINGS

rhel-registration	rhsm / RhsmVars
rhel_reg_method	**rhsm_method**
rhel_reg_org	**rhsm_org_id**
rhel_reg_pool_id	**rhsm_pool_ids**
rhel_reg_activation_key	**rhsm_activation_key**
rhel_reg_auto_attach	**rhsm_autosubscribe**
rhel_reg_sat_url	**rhsm_satellite_url**
rhel_reg_repos	**rhsm_repos**
rhel_reg_user	**rhsm_username**

rhel-registration	rhsm / RhsmVars
rhel_reg_password	**rhsm_password**
rhel_reg_http_proxy_host	**rhsm_rhsm_proxy_hostname**
rhel_reg_http_proxy_port	**rhsm_rhsm_proxy_port**
rhel_reg_http_proxy_username	**rhsm_rhsm_proxy_user**
rhel_reg_http_proxy_password	**rhsm_rhsm_proxy_password**

Now that you have configured the environment file for the **rhsm** service, you can include it with your next overcloud deployment operation.

5.7. DEPLOYING THE OVERCLOUD WITH THE RHSM COMPOSABLE SERVICE

This section shows how to apply your **rhsm** configuration to the overcloud.

Procedure

1. Include **rhsm.yml** environment file with the **openstack overcloud deploy**:

```
openstack overcloud deploy \
  <other cli args> \
  -e ~/templates/rhsm.yaml
```

   This enables the Ansible configuration of the overcloud and the Ansible-based registration.

2. Wait until the overcloud deployment completes.

3. Check the subscription details on your overcloud nodes. For example, log into a Controller node and run the following commands:

```
$ sudo subscription-manager status
$ sudo subscription-manager list --consumed
```

In addition to the director-based registration method, you can also manually register after deployment.

5.8. RUNNING ANSIBLE-BASED REGISTRATION MANUALLY

You can perform manual Ansible-based registration on a deployed overcloud. You accomplish this using the director's dynamic inventory script to define node roles as host groups and then run a playbook against them using **ansible-playbook**. The following example shows how to manually register Controller nodes using a playbook.

Procedure

1. Create a playbook with that using the **redhat_subscription** modules to register your nodes. For example, the following playbook applies to Controller nodes:

```
---
- name: Register Controller nodes
  hosts: Controller
  become: yes
  vars:
    repos:
      - rhel-8-for-x86_64-baseos-rpms
      - rhel-8-for-x86_64-appstream-rpms
      - rhel-8-for-x86_64-highavailability-rpms
      - ansible-2.8-for-rhel-8-x86_64-rpms
      - openstack-beta-for-rhel-8-x86_64-rpms
      - rhceph-4-mon-for-rhel-8-x86_64-rpms
  tasks:
    - name: Register system
      redhat_subscription:
        username: myusername
        password: p@55w0rd!
        org_id: 1234567
        pool_ids: 1a85f9223e3d5e43013e3d6e8ff506fd
    - name: Disable all repos
      command: "subscription-manager repos --disable *"
    - name: Enable Controller node repos
      command: "subscription-manager repos --enable {{ item }}"
      with_items: "{{ repos }}"
```

   - This play contains three tasks:
     - Register the node using an activation key.
     - Disable any auto-enabled repositories.
     - Enable only the repositories relevant to the Controller node. The repositories are listed with the **repos** variable.

2. After deploying the overcloud, you can run the following command so that Ansible executes the playbook (**ansible-osp-registration.yml**) against your overcloud:

```
$ ansible-playbook -i /usr/bin/tripleo-ansible-inventory ansible-osp-registration.yml
```

   This command does the following:

   - Runs the dynamic inventory script to get a list of host and their groups.
   - Applies the playbook tasks to the nodes in the group defined in the playbook's **hosts** parameter, which in this case is the Controller group.

CHAPTER 6. COMPOSABLE SERVICES AND CUSTOM ROLES

The Overcloud usually consists of nodes in predefined roles such as Controller nodes, Compute nodes, and different storage node types. Each of these default roles contains a set of services defined in the core Heat template collection on the director node. However, the architecture of the core Heat templates provide methods to do the following tasks:

- Create custom roles
- Add and remove services from each role

This allows the possibility to create different combinations of services on different roles. This chapter explores the architecture of custom roles, composable services, and methods for using them.

6.1. SUPPORTED ROLE ARCHITECTURE

The following architectures are available when using custom roles and composable services:

Architecture 1 - Default Architecture

Uses the default **roles_data** files. All controller services are contained within one Controller role.

Architecture 2 - Supported Standalone Roles

Use the predefined files in **/usr/share/openstack-tripleo-heat-templates/roles** to generate a custom **roles_data** file`.

Architecture 3 - Custom Composable Services

Create your own **roles** and use them to generate a custom **roles_data** file. Note that only a limited number of composable service combinations have been tested and verified and Red Hat cannot support all composable service combinations.

6.2. ROLES

6.2.1. Examining the roles_data File

The Overcloud creation process defines its roles using a **roles_data** file. The **roles_data** file contains a YAML-formatted list of the roles. The following is a shortened example of the **roles_data** syntax:

```
- name: Controller
  description: |
    Controller role that has all the controller services loaded and handles
    Database, Messaging and Network functions.
  ServicesDefault:
    - OS::TripleO::Services::AuditD
    - OS::TripleO::Services::CACerts
    - OS::TripleO::Services::CephClient
    ...
- name: Compute
  description: |
    Basic Compute Node role
  ServicesDefault:
    - OS::TripleO::Services::AuditD
    - OS::TripleO::Services::CACerts
    - OS::TripleO::Services::CephClient
    ...
```

The core Heat template collection contains a default **roles_data** file located at **/usr/share/openstack-tripleo-heat-templates/roles_data.yaml**. The default file defines the following role types:

- **Controller**
- **Compute**
- **BlockStorage**
- **ObjectStorage**
- **CephStorage**.

The **openstack overcloud deploy** command includes this file during deployment. You can override this file with a custom **roles_data** file using the **-r** argument. For example:

```
$ openstack overcloud deploy --templates -r ~/templates/roles_data-custom.yaml
```

6.2.2. Creating a roles_data File

Although you can manually create a custom **roles_data** file, you can also automatically generating the file using individual role templates. The director provides several commands to manage role templates and automatically generate a custom **roles_data** file.

To list the default role templates, use the **openstack overcloud roles list** command:

```
$ openstack overcloud roles list
BlockStorage
CephStorage
Compute
ComputeHCI
ComputeOvsDpdk
Controller
...
```

To see the role's YAML definition, use the **openstack overcloud roles show** command:

```
$ openstack overcloud roles show Compute
```

To generate a custom **roles_data** file, use the **openstack overcloud roles generate** command to join multiple predefined roles into a single file. For example, the following command joins the **Controller**, **Compute**, and **Networker** roles into a single file:

```
$ openstack overcloud roles generate -o ~/roles_data.yaml Controller Compute Networker
```

The **-o** defines the name of the file to create.

This creates a custom **roles_data** file. However, the previous example uses the **Controller** and **Networker** roles, which both contain the same networking agents. This means the networking services scale from **Controller** to the **Networker** role. The overcloud balances the load for networking services between the **Controller** and **Networker** nodes.

To make this **Networker** role standalone, you can create your own custom **Controller** role, as well as any other role needed. This allows you to generate a **roles_data** file from your own custom roles.

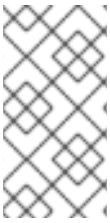
Copy the directory from the core Heat template collection to the **stack** user's home directory:

```
$ cp -r /usr/share/openstack-tripleo-heat-templates/roles ~/.
```

Add or modify the custom role files in this directory. Use the **--roles-path** option with any of the aforementioned role sub-commands to use this directory as the source for your custom roles. For example:

```
$ openstack overcloud roles generate -o my_roles_data.yaml \
  --roles-path ~/roles \
  Controller Compute Networker
```

This generates a single **my_roles_data.yaml** file from the individual roles in the **~/roles** directory.



NOTE

The default roles collection also contains the **ControllerOpenStack** role, which does not include services for **Networker**, **Messaging**, and **Database** roles. You can use the **ControllerOpenStack** combined with with the standalone **Networker**, **Messaging**, and **Database** roles.

6.2.3. Supported Custom Roles

The following table contains information about the available custom roles. You can find custom role templates in the **/usr/share/openstack-tripleo-heat-templates/roles** directory.

Role	Description	File
BlockStorage	OpenStack Block Storage (cinder) node.	**BlockStorage.yaml**
CephAll	Full standalone Ceph Storage node. Includes OSD, MON, Object Gateway (RGW), Object Operations (MDS), Manager (MGR), and RBD Mirroring.	**CephAll.yaml**
CephFile	Standalone scale-out Ceph Storage file role. Includes OSD and Object Operations (MDS).	**CephFile.yaml**
CephObject	Standalone scale-out Ceph Storage object role. Includes OSD and Object Gateway (RGW).	**CephObject.yaml**
CephStorage	Ceph Storage OSD node role.	**CephStorage.yaml**
ComputeAlt	Alternate Compute node role.	**ComputeAlt.yaml**
ComputeDVR	DVR enabled Compute node role.	**ComputeDVR.yaml**
ComputeHCI	Compute node with hyper-converged infrastructure. Includes Compute and Ceph OSD services.	**ComputeHCI.yaml**

Role	Description	File
ComputeInstanceHA	Compute Instance HA node role. Use in conjunction with the **environments/compute-instanceha.yaml`** environment file.	**ComputeInstanceHA.yaml**
ComputeLiquidio	Compute node with Cavium Liquidio Smart NIC.	**ComputeLiquidio.yaml**
ComputeOvsDpdkRT	Compute OVS DPDK RealTime role.	**ComputeOvsDpdkRT.yaml**
ComputeOvsDpdk	Compute OVS DPDK role.	**ComputeOvsDpdk.yaml**
ComputePPC64LE	Compute role for ppc64le servers.	**ComputePPC64LE.yaml**
ComputeRealTime	Compute role optimized for real-time behaviour. When using this role, it is mandatory that an **overcloud-realtime-compute** image is available and the role specific parameters **IsolCpusList** and **NovaVcpuPinSet** are set accordingly to the hardware of the real-time compute nodes.	**ComputeRealTime.yaml**
ComputeSriovRT	Compute SR-IOV RealTime role.	**ComputeSriovRT.yaml**
ComputeSriov	Compute SR-IOV role.	**ComputeSriov.yaml**
Compute	Standard Compute node role.	**Compute.yaml**
ControllerAllNovaStandalone	Controller role that does not contain the database, messaging, networking, and OpenStack Compute (nova) control components. Use in combination with the **Database**, **Messaging**, **Networker**, and **Novacontrol** roles.	**ControllerAllNovaStandalone.yaml**
ControllerNoCeph	Controller role with core Controller services loaded but no Ceph Storage (MON) components. This role handles database, messaging, and network functions but not any Ceph Storage functions.	**ControllerNoCeph.yaml**
ControllerNovaStandalone	Controller role that does not contain the OpenStack Compute (nova) control component. Use in combination with the **Novacontrol** role.	**ControllerNovaStandalone.yaml**

Role	Description	File
ControllerOpenstack	Controller role that does not contain the database, messaging, and networking components. Use in combination with the **Database**, **Messaging**, and **Networker** roles.	**ControllerOpenstack.yaml**
ControllerStorageNfs	Controller role with all core services loaded and uses Ceph NFS. This roles handles database, messaging, and network functions.	**ControllerStorageNfs.yaml**
Controller	Controller role with all core services loaded. This roles handles database, messaging, and network functions.	**Controller.yaml**
Database	Standalone database role. Database managed as a Galera cluster using Pacemaker.	**Database.yaml**
HciCephAll	Compute node with hyper-converged infrastructure and all Ceph Storage services. Includes OSD, MON, Object Gateway (RGW), Object Operations (MDS), Manager (MGR), and RBD Mirroring.	**HciCephAll.yaml**
HciCephFile	Compute node with hyper-converged infrastructure and Ceph Storage file services. Includes OSD and Object Operations (MDS).	**HciCephFile.yaml**
HciCephMon	Compute node with hyper-converged infrastructure and Ceph Storage block services. Includes OSD, MON, and Manager.	**HciCephMon.yaml**
HciCephObject	Compute node with hyper-converged infrastructure and Ceph Storage object services. Includes OSD and Object Gateway (RGW).	**HciCephObject.yaml**
IronicConductor	Ironic Conductor node role.	**IronicConductor.yaml**
Messaging	Standalone messaging role. RabbitMQ managed with Pacemaker.	**Messaging.yaml**
Networker	Standalone networking role. Runs OpenStack networking (neutron) agents on their own.	**Networker.yaml**
Novacontrol	Standalone **nova-control** role to run OpenStack Compute (nova) control agents on their own.	**Novacontrol.yaml**
ObjectStorage	Swift Object Storage node role.	**ObjectStorage.yaml**
Telemetry	Telemetry role with all the metrics and alarming services.	**Telemetry.yaml**

6.2.4. Examining Role Parameters

Each role uses the following parameters:

name

(Mandatory) The name of the role, which is a plain text name with no spaces or special characters. Check that the chosen name does not cause conflicts with other resources. For example, use **Networker** as a name instead of **Network**.

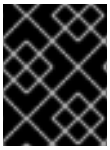
description

(Optional) A plain text description for the role.

tags

(Optional) A YAML list of tags that define role properties. Use this parameter to define the primary role with both the **controller** and **primary** tags together:

```
- name: Controller
  ...
  tags:
    - primary
    - controller
  ...
```



IMPORTANT

If you do not tag the primary role, the first role defined becomes the primary role. Ensure that this role is the Controller role.

networks

A list of networks to configure on the role. Default networks include **External**, **InternalApi**, **Storage**, **StorageMgmt**, **Tenant**, and **Management**.

CountDefault

(Optional) Defines the default number of nodes to deploy for this role.

HostnameFormatDefault

(Optional) Defines the default hostname format for the role. The default naming convention uses the following format:

```
[STACK NAME]-[ROLE NAME]-[NODE ID]
```

For example, the default Controller nodes are named:

```
overcloud-controller-0
overcloud-controller-1
overcloud-controller-2
...
```

disable_constraints

(Optional) Defines whether to disable OpenStack Compute (nova) and OpenStack Image Storage (glance) constraints when deploying with the director. Used when deploying an overcloud with pre-provisioned nodes. For more information, see "Configuring a Basic Overcloud using Pre-Provisioned Nodes" in the *Director Installation and Usage Guide*.

disable_upgrade_deployment

(Optional) Defines whether to disable upgrades for a specific role. This provides a method to upgrade individual nodes in a role and ensure availability of services. For example, the Compute and Swift Storage roles use this parameter.

upgrade_batch_size

(Optional) Defines the number of tasks to execute in a batch during the upgrade. A task counts as one upgrade step per node. The default batch size is 1, which means the upgrade process executes a single upgrade step on each node, one at a time. Increasing the batch size increases the number of tasks executed simultaneously on nodes

ServicesDefault

(Optional) Defines the default list of services to include on the node. See Section 6.3.2, "Examining Composable Service Architecture" for more information.

These parameters provide a means to create new roles and also define which services to include.

The **openstack overcloud deploy** command integrates the parameters from the **roles_data** file into some of the Jinja2-based templates. For example, at certain points, the **overcloud.j2.yaml** Heat template iterates over the list of roles from **roles_data.yaml** and creates parameters and resources specific to each respective role.

The resource definition for each role in the **overcloud.j2.yaml** Heat template appears as the following snippet:

```
  {{role.name}}:
    type: OS::Heat::ResourceGroup
    depends_on: Networks
    properties:
      count: {get_param: {{role.name}}Count}
      removal_policies: {get_param: {{role.name}}RemovalPolicies}
      resource_def:
        type: OS::TripleO::{{role.name}}
        properties:
          CloudDomain: {get_param: CloudDomain}
          ServiceNetMap: {get_attr: [ServiceNetMap, service_net_map]}
          EndpointMap: {get_attr: [EndpointMap, endpoint_map]}
...
```

This snippet shows how the Jinja2-based template incorporates the **{{role.name}}** variable to define the name of each role as a **OS::Heat::ResourceGroup** resource. This in turn uses each **name** parameter from the **roles_data** file to name each respective **OS::Heat::ResourceGroup** resource.

6.2.5. Creating a New Role

In this example, the aim is to create a new **Horizon** role to host the OpenStack Dashboard (**horizon**) only. In this situation, you create a custom **roles** directory that includes the new role information.

Create a custom copy of the default **roles** directory:

```
$ cp -r /usr/share/openstack-tripleo-heat-templates/roles ~/.
```

Create a new file called **~/roles/Horizon.yaml** and create a new **Horizon** role containing base and core OpenStack Dashboard services. For example:

```
- name: Horizon
  CountDefault: 1
  HostnameFormatDefault: '%stackname%-horizon-%index%'
  ServicesDefault:
    - OS::TripleO::Services::CACerts
    - OS::TripleO::Services::Kernel
    - OS::TripleO::Services::Ntp
    - OS::TripleO::Services::Snmp
    - OS::TripleO::Services::Sshd
    - OS::TripleO::Services::Timezone
    - OS::TripleO::Services::TripleoPackages
    - OS::TripleO::Services::TripleoFirewall
    - OS::TripleO::Services::SensuClient
    - OS::TripleO::Services::FluentdClient
    - OS::TripleO::Services::AuditD
    - OS::TripleO::Services::Collectd
    - OS::TripleO::Services::MySQLClient
    - OS::TripleO::Services::Apache
    - OS::TripleO::Services::Horizon
```

It is a good idea to set the **CountDefault** to **1** so that a default Overcloud always includes the **Horizon** node.

If scaling the services in an existing overcloud, keep the existing services on the **Controller** role. If creating a new overcloud and you want the OpenStack Dashboard to remain on the standalone role, remove the OpenStack Dashboard components from the **Controller** role definition:

```
- name: Controller
  CountDefault: 1
  ServicesDefault:
    ...
    - OS::TripleO::Services::GnocchiMetricd
    - OS::TripleO::Services::GnocchiStatsd
    - OS::TripleO::Services::HAproxy
    - OS::TripleO::Services::HeatApi
    - OS::TripleO::Services::HeatApiCfn
    - OS::TripleO::Services::HeatApiCloudwatch
    - OS::TripleO::Services::HeatEngine
    # - OS::TripleO::Services::Horizon              # Remove this service
    - OS::TripleO::Services::IronicApi
    - OS::TripleO::Services::IronicConductor
    - OS::TripleO::Services::Iscsid
    - OS::TripleO::Services::Keepalived
    ...
```

Generate the new **roles_data** file using the **roles** directory as the source:

```
$ openstack overcloud roles generate -o roles_data-horizon.yaml \
  --roles-path ~/roles \
  Controller Compute Horizon
```

You might need to define a new flavor for this role so that you can tag specific nodes. For this example, use the following commands to create a **horizon** flavor:

```
$ openstack flavor create --id auto --ram 6144 --disk 40 --vcpus 4 horizon
```

```
$ openstack flavor set --property "cpu_arch"="x86_64" --property "capabilities:boot_option"="local" --
property "capabilities:profile"="horizon" horizon
$ openstack flavor set --property resources:VCPU=0 --property resources:MEMORY_MB=0 --
property resources:DISK_GB=0 --property resources:CUSTOM_BAREMETAL=1 horizon
```

Tag nodes into the new flavor using the following command:

```
$ openstack baremetal node set --property capabilities='profile:horizon,boot_option:local' 58c3d07e-
24f2-48a7-bbb6-6843f0e8ee13
```

Define the Horizon node count and flavor using the following environment file snippet:

```
parameter_defaults:
  OvercloudHorizonFlavor: horizon
  HorizonCount: 1
```

Include the new **roles_data** file and environment file when running the **openstack overcloud deploy** command. For example:

```
$ openstack overcloud deploy --templates -r ~/templates/roles_data-horizon.yaml -e
~/templates/node-count-flavor.yaml
```

When the deployment completes, this creates a three-node Overcloud consisting of one Controller node, one Compute node, and one Networker node. To view the Overcloud's list of nodes, run the following command:

```
$ openstack server list
```

6.3. COMPOSABLE SERVICES

6.3.1. Guidelines and Limitations

Note the following guidelines and limitations for the composable node architecture.

For services not managed by Pacemaker:

- You can assign services to standalone custom roles.
- You can create additional custom roles after the initial deployment and deploy them to scale existing services.

For services managed by Pacemaker:

- You can assign Pacemaker-managed services to standalone custom roles.
- Pacemaker has a 16 node limit. If you assign the Pacemaker service (**OS::TripleO::Services::Pacemaker**) to 16 nodes, subsequent nodes must use the Pacemaker Remote service (**OS::TripleO::Services::PacemakerRemote**) instead. You cannot have the Pacemaker service and Pacemaker Remote service on the same role.
- Do not include the Pacemaker service (**OS::TripleO::Services::Pacemaker**) on roles that do not contain Pacemaker-managed services.

- You cannot scale up or scale down a custom role that contains **OS::TripleO::Services::Pacemaker** or **OS::TripleO::Services::PacemakerRemote** services.

General limitations:

- You cannot change custom roles and composable services during the a major version upgrade.
- You cannot modify the list of services for any role after deploying an Overcloud. Modifying the service lists after Overcloud deployment can cause deployment errors and leave orphaned services on nodes.

6.3.2. Examining Composable Service Architecture

The core Heat template collection contains two sets of composable service templates:

- **deployment** contains the templates for key OpenStack Platform services.
- **puppet/services** contains legacy templates for configuring composable services. In some cases, the composable services use templates from this directory for compatibility. In most cases, the composable services use the templates in the **deployment** directory.

Each template contains a description that identifies its purpose. For example, the **deployment/time/ntp-baremetal-puppet.yaml** service template contains the following description:

```
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.
```

These service templates are registered as resources specific to a Red Hat OpenStack Platform deployment. This means you can call each resource using a unique Heat resource namespace defined in the **overcloud-resource-registry-puppet.j2.yaml** file. All services use the **OS::TripleO::Services** namespace for their resource type.

Some resources use the base composable service templates directly. For example:

```
resource_registry:
  ...
  OS::TripleO::Services::Ntp: deployment/time/ntp-baremetal-puppet.yaml
  ...
```

However, core services require containers and use the containerized service templates. For example, the **keystone** containerized service uses the following resource:

```
resource_registry:
  ...
  OS::TripleO::Services::Keystone: deployment/keystone/keystone-container-puppet.yaml
  ...
```

These containerized templates usually reference other templates to include dependencies. For example, the **deployment/keystone/keystone-container-puppet.yaml** template stores the output of the base template in the **ContainersCommon** resource:

```
resources:
  ContainersCommon:
    type: ../containers-common.yaml
```

The containerized template can then incorporate functions and data from the **containers-common.yaml** template.

The **overcloud.j2.yaml** Heat template includes a section of Jinja2-based code to define a service list for each custom role in the **roles_data.yaml** file:

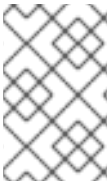
```
{{role.name}}Services:
  description: A list of service resources (configured in the Heat
               resource_registry) which represent nested stacks
               for each service that should get installed on the {{role.name}} role.
  type: comma_delimited_list
  default: {{role.ServicesDefault|default([])}}
```

For the default roles, this creates the following service list parameters: **ControllerServices**, **ComputeServices**, **BlockStorageServices**, **ObjectStorageServices**, and **CephStorageServices**.

You define the default services for each custom role in the **roles_data.yaml** file. For example, the default Controller role contains the following content:

```
- name: Controller
  CountDefault: 1
  ServicesDefault:
    - OS::TripleO::Services::CACerts
    - OS::TripleO::Services::CephMon
    - OS::TripleO::Services::CephExternal
    - OS::TripleO::Services::CephRgw
    - OS::TripleO::Services::CinderApi
    - OS::TripleO::Services::CinderBackup
    - OS::TripleO::Services::CinderScheduler
    - OS::TripleO::Services::CinderVolume
    - OS::TripleO::Services::Core
    - OS::TripleO::Services::Kernel
    - OS::TripleO::Services::Keystone
    - OS::TripleO::Services::GlanceApi
    - OS::TripleO::Services::GlanceRegistry
...
```

These services are then defined as the default list for the **ControllerServices** parameter.



NOTE

You can also use an environment file to override the default list for the service parameters. For example, you can define **ControllerServices** as a **parameter_default** in an environment file to override the services list from the **roles_data.yaml** file.

6.3.3. Adding and Removing Services from Roles

The basic method of adding or removing services involves creating a copy of the default service list for a node role and then adding or removing services. For example, you might aim to remove OpenStack Orchestration (**heat**) from the Controller nodes. In this situation, create a custom copy of the default

roles directory:

```
$ cp -r /usr/share/openstack-tripleo-heat-templates/roles ~/.
```

Edit the **~/roles/Controller.yaml** file and modify the service list for the **ServicesDefault** parameter. Scroll to the OpenStack Orchestration services and remove them:

```
    - OS::TripleO::Services::GlanceApi
    - OS::TripleO::Services::GlanceRegistry
    - OS::TripleO::Services::HeatApi            # Remove this service
    - OS::TripleO::Services::HeatApiCfn         # Remove this service
    - OS::TripleO::Services::HeatApiCloudwatch  # Remove this service
    - OS::TripleO::Services::HeatEngine         # Remove this service
    - OS::TripleO::Services::MySQL
    - OS::TripleO::Services::NeutronDhcpAgent
```

Generate the new **roles_data** file. For example:

```
$ openstack overcloud roles generate -o roles_data-no_heat.yaml \
  --roles-path ~/roles \
  Controller Compute Networker
```

Include this new **roles_data** file when running the **openstack overcloud deploy** command. For example:

```
$ openstack overcloud deploy --templates -r ~/templates/roles_data-no_heat.yaml
```

This deploys an Overcloud without OpenStack Orchestration services installed on the Controller nodes.

NOTE

You can also disable services in the **roles_data** file using a custom environment file. Redirect the services to disable to the **OS::Heat::None** resource. For example:

```
resource_registry:
  OS::TripleO::Services::HeatApi: OS::Heat::None
  OS::TripleO::Services::HeatApiCfn: OS::Heat::None
  OS::TripleO::Services::HeatApiCloudwatch: OS::Heat::None
  OS::TripleO::Services::HeatEngine: OS::Heat::None
```

6.3.4. Enabling Disabled Services

Some services are disabled by default. These services are registered as null operations (**OS::Heat::None**) in the **overcloud-resource-registry-puppet.j2.yaml** file. For example, the Block Storage backup service (**cinder-backup**) is disabled:

```
  OS::TripleO::Services::CinderBackup: OS::Heat::None
```

To enable this service, include an environment file that links the resource to its respective Heat templates in the **puppet/services** directory. Some services have predefined environment files in the **environments** directory. For example, the Block Storage backup service uses the **environments/cinder-backup.yaml** file, which contains the following:

-

```
resource_registry:
  OS::TripleO::Services::CinderBackup: ../puppet/services/pacemaker/cinder-backup.yaml
...
```

This overrides the default null operation resource and enables the service. Include this environment file when running the **openstack overcloud deploy** command.

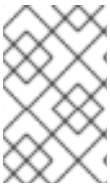
```
$ openstack overcloud deploy --templates -e /usr/share/openstack-tripleo-heat-
templates/environments/cinder-backup.yaml
```

TIP

For another example of how to enable disabled services, see the Installation section of the OpenStack Data Processing guide. This section contains instructions on how to enable the OpenStack Data Processing service (**sahara**) on the overcloud.

6.3.5. Creating a Generic Node with No Services

Red Hat OpenStack Platform provides the ability to create generic Red Hat Enterprise Linux 8 nodes without any OpenStack services configured. This is useful when you need to host software outside of the core Red Hat OpenStack Platform environment. For example, OpenStack Platform provides integration with monitoring tools such as Kibana and Sensu (see Monitoring Tools Configuration Guide). While Red Hat does not provide support for the monitoring tools themselves, the director can create a generic Red Hat Enterprise Linux 8 node to host these tools.



NOTE

The generic node still uses the base **overcloud-full** image rather than a base Red Hat Enterprise Linux 8 image. This means the node has some Red Hat OpenStack Platform software installed but not enabled or configured.

Creating a generic node requires a new role without a **ServicesDefault** list:

```
- name: Generic
```

Include the role in your custom **roles_data** file (**roles_data_with_generic.yaml**). Make sure to keep the existing **Controller** and **Compute** roles.

You can also include an environment file (**generic-node-params.yaml**) to specify how many generic Red Hat Enterprise Linux 8 nodes you require and the flavor when selecting nodes to provision. For example:

```
parameter_defaults:
  OvercloudGenericFlavor: baremetal
  GenericCount: 1
```

Include both the roles file and the environment file when running the **openstack overcloud deploy** command. For example:

```
$ openstack overcloud deploy --templates -r ~/templates/roles_data_with_generic.yaml -e
~/templates/generic-node-params.yaml
```

This deploys a three-node environment with one Controller node, one Compute node, and one generic Red Hat Enterprise Linux 8 node.

CHAPTER 7. CONTAINERIZED SERVICES

The director installs the core OpenStack Platform services as containers on the overcloud. This section provides some background information on how containerized services work.

7.1. CONTAINERIZED SERVICE ARCHITECTURE

The director installs the core OpenStack Platform services as containers on the overcloud. The templates for the containerized services are located in the **/usr/share/openstack-tripleo-heat-templates/deployment/**.

All nodes using containerized services must enable the **OS::TripleO::Services::Podman** service. When you create a **roles_data.yaml** file for your custom roles configuration, include the **OS::TripleO::Services::Podman** service with the base composable services, as the containerized services. For example, the **IronicConductor** role uses the following role definition:

```
- name: IronicConductor
  description: |
    Ironic Conductor node role
  networks:
    InternalApi:
      subnet: internal_api_subnet
    Storage:
      subnet: storage_subnet
  HostnameFormatDefault: '%stackname%-ironic-%index%'
  ServicesDefault:
    - OS::TripleO::Services::Aide
    - OS::TripleO::Services::AuditD
    - OS::TripleO::Services::BootParams
    - OS::TripleO::Services::CACerts
    - OS::TripleO::Services::CertmongerUser
    - OS::TripleO::Services::Collectd
    - OS::TripleO::Services::Docker
    - OS::TripleO::Services::Fluentd
    - OS::TripleO::Services::IpaClient
    - OS::TripleO::Services::Ipsec
    - OS::TripleO::Services::IronicConductor
    - OS::TripleO::Services::IronicPxe
    - OS::TripleO::Services::Kernel
    - OS::TripleO::Services::LoginDefs
    - OS::TripleO::Services::MetricsQdr
    - OS::TripleO::Services::MySQLClient
    - OS::TripleO::Services::ContainersLogrotateCrond
    - OS::TripleO::Services::Podman
    - OS::TripleO::Services::Rhsm
    - OS::TripleO::Services::SensuClient
    - OS::TripleO::Services::Snmp
    - OS::TripleO::Services::Timesync
    - OS::TripleO::Services::Timezone
    - OS::TripleO::Services::TripleoFirewall
    - OS::TripleO::Services::TripleoPackages
    - OS::TripleO::Services::Tuned
```

7.2. CONTAINERIZED SERVICE PARAMETERS

Each containerized service template contains an **outputs** section that defines a data set passed to the director's OpenStack Orchestration (Heat) service. In addition to the standard composable service parameters (see Section 6.2.4, "Examining Role Parameters"), the template contain a set of parameters specific to the container configuration.

puppet_config

Data to pass to Puppet when configuring the service. In the initial overcloud deployment steps, the director creates a set of containers used to configure the service before the actual containerized service runs. This parameter includes the following sub-parameters: +

- **config_volume** - The mounted volume that stores the configuration.
- **puppet_tags** - Tags to pass to Puppet during configuration. These tags are used in OpenStack Platform to restrict the Puppet run to a particular service's configuration resource. For example, the OpenStack Identity (keystone) containerized service uses the **keystone_config** tag to ensure that all require only the **keystone_config** Puppet resource run on the configuration container.
- **step_config** - The configuration data passed to Puppet. This is usually inherited from the referenced composable service.
- **config_image** - The container image used to configure the service.

kolla_config

A set of container-specific data that defines configuration file locations, directory permissions, and the command to run on the container to launch the service.

docker_config

Tasks to run on the service's configuration container. All tasks are grouped into the following steps to help the director perform a staged deployment:

- **Step 1** - Load balancer configuration
- **Step 2** - Core services (Database, Redis)
- **Step 3** - Initial configuration of OpenStack Platform service
- **Step 4** - General OpenStack Platform services configuration
- **Step 5** - Service activation

host_prep_tasks

Preparation tasks for the bare metal node to accommodate the containerized service.

7.3. PREPARING CONTAINER IMAGES

The overcloud configuration requires initial registry configuration to determine where to obtain images and how to store them. Complete the following steps to generate and customize an environment file for preparing your container images.

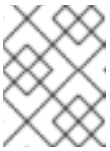
Procedure

1. Log in to your undercloud host as the stack user.
2. Generate the default container image preparation file:

```
$ openstack tripleo container image prepare default \
  --local-push-destination \
  --output-env-file containers-prepare-parameter.yaml
```

This command includes the following additional options:

- **--local-push-destination** sets the registry on the undercloud as the location for container images. This means the director pulls the necessary images from the Red Hat Container Catalog and pushes them to the registry on the undercloud. The director uses this registry as the container image source. To pull directly from the Red Hat Container Catalog, omit this option.
- **--output-env-file** is an environment file name. The contents of this file include the parameters for preparing your container images. In this case, the name of the file is **containers-prepare-parameter.yaml**.



NOTE

You can also use the same **containers-prepare-parameter.yaml** file to define a container image source for both the undercloud and the overcloud.

3. Edit the **containers-prepare-parameter.yaml** and make the modifications to suit your requirements.

7.4. CONTAINER IMAGE PREPARATION PARAMETERS

The default file for preparing your containers (**containers-prepare-parameter.yaml**) contains the **ContainerImagePrepare** Heat parameter. This parameter defines a list of strategies for preparing a set of images:

```
parameter_defaults:
  ContainerImagePrepare:
  - (strategy one)
  - (strategy two)
  - (strategy three)
  ...
```

Each strategy accepts a set of sub-parameters that define which images to use and what to do with them. The following table contains information about the sub-parameters you can use with each **ContainerImagePrepare** strategy:

Parameter	Description
excludes	List of image name substrings to exclude from a strategy.
includes	List of image name substrings to include in a strategy. At least one image name must match an existing image. All **excludes** are ignored if **includes** is specified.

Parameter	Description
modify_append_tag	String to append to the tag for the destination image. For example, if you pull an image with the tag **14.0-89** and set the **modify_append_tag** to **-hotfix**, the director tags the final image as **14.0-89-hotfix**.
modify_only_with_labels	A dictionary of image labels that filter the images to modify. If an image matches the labels defined, the director includes the image in the modification process.
modify_role	String of ansible role names to run during upload but before pushing the image to the destination registry.
modify_vars	Dictionary of variables to pass to **modify_role**.
push_destination	The namespace of the registry to push images during the upload process. When you specify a namespace for this parameter, all image parameters use this namespace too. If set to **true**, the **push_destination** is set to the undercloud registry namespace. It is not recommended to set this parameters to **false** in production environments.
pull_source	The source registry from where to pull the original container images.
set	A dictionary of **key: value** definitions that define where to obtain the initial images.
tag_from_label	Defines the label pattern to tag the resulting images. Usually sets to **{version}-{release}**.

The **set** parameter accepts a set of **key: value** definitions. The following table contains information about the keys:

Key	Description
ceph_image	The name of the Ceph Storage container image.
ceph_namespace	The namespace of the Ceph Storage container image.
ceph_tag	The tag of the Ceph Storage container image.
name_prefix	A prefix for each OpenStack service image.

Key	Description
name_suffix	A suffix for each OpenStack service image.
namespace	The namespace for each OpenStack service image.
neutron_driver	The driver to use to determine which OpenStack Networking (neutron) container to use. Use a null value to set to the standard **neutron-server** container. Set to **ovn** to use OVN-based containers. Set to **odl** to use OpenDaylight-based containers.
tag	The tag that the director uses to identify the images to pull from the source registry. You usually keep this key set to **latest**.

The **ContainerImageRegistryCredentials** parameter maps a container registry to a username and password to authenticate to that registry.

If a container registry requires a username and password, you can use **ContainerImageRegistryCredentials** to include their values with the following syntax:

```
ContainerImagePrepare:
- push_destination: 192.168.24.1:8787
  set:
    namespace: registry.redhat.io/...
    ...
ContainerImageRegistryCredentials:
  registry.redhat.io:
    my_username: my_password
```

In the example, replace **my_username** and **my_password** with the credentials you use to access content from the Red Hat Content Distribution Network.

7.5. LAYERING IMAGE PREPARATION ENTRIES

The value of the **ContainerImagePrepare** parameter is a YAML list. This means you can specify multiple entries. The following example demonstrates two entries where the director uses the latest version of all images except for the **nova-api** image, which uses the version tagged with **15.0-44**:

```
ContainerImagePrepare:
- tag_from_label: "{version}-{release}"
  push_destination: true
  excludes:
  - nova-api
  set:
    namespace: registry.access.redhat.com/rhosp15
    name_prefix: openstack-
    name_suffix: ''
    tag: latest
- push_destination: true
  includes:
```

```
    - nova-api
  set:
    namespace: registry.access.redhat.com/rhosp15
    tag: 15.0-44
```

The **includes** and **excludes** entries control image filtering for each entry. The images that match the **includes** strategy take precedence over **excludes** matches. The image name must include the **includes** or **excludes** value to be considered a match.

7.6. MODIFYING IMAGES DURING PREPARATION

It is possible to modify images during image preparation, then immediately deploy with modified images. Scenarios for modifying images include:

- As part of a continuous integration pipeline where images are modified with the changes being tested before deployment.
- As part of a development workflow where local changes need to be deployed for testing and development.
- When changes need to be deployed but are not available through an image build pipeline. For example, adding proprietry add-ons or emergency fixes.

To modify an image during preparation, invoke an Ansible role on each image that you want to modify. The role takes a source image, makes the requested changes, and tags the result. The prepare command can push the image to the destination registry and set the Heat parameters to refer to the modified image.

The Ansible role **tripleo-modify-image** conforms with the required role interface, and provides the behaviour necessary for the modify use-cases. Modification is controlled using modify-specific keys in the **ContainerImagePrepare** parameter:

- **modify_role** specifies the Ansible role to invoke for each image to modify.
- **modify_append_tag** appends a string to the end of the source image tag. This makes it obvious that the resulting image has been modified. Use this parameter to skip modification if the **push_destination** registry already contains the modified image. It is recommended to change **modify_append_tag** whenever you modify the image.
- **modify_vars** is a dictionary of Ansible variables to pass to the role.

To select a use-case that the **tripleo-modify-image** role handles, set the **tasks_from** variable to the required file in that role.

While developing and testing the **ContainerImagePrepare** entries that modify images, it is recommended to run the image prepare command without any additional options to confirm the image is modified as expected:

```
sudo openstack tripleo container image prepare \
  -e ~/containers-prepare-parameter.yaml
```

7.7. UPDATING EXISTING PACKAGES ON CONTAINER IMAGES

The following example **ContainerImagePrepare** entry updates in all packages on the images using the undercloud host's yum repository configuration:

```
ContainerImagePrepare:
- push_destination: true
  ...
  modify_role: tripleo-modify-image
  modify_append_tag: "-updated"
  modify_vars:
    tasks_from: yum_update.yml
    compare_host_packages: true
    yum_repos_dir_path: /etc/yum.repos.d
  ...
```

7.8. INSTALLING ADDITIONAL RPM FILES TO CONTAINER IMAGES

You can install a directory of RPM files in your container images. This is useful for installing hotfixes, local package builds, or any package not available through a package repository. For example, the following **ContainerImagePrepare** entry installs some hotfix packages only on the **nova-compute** image:

```
ContainerImagePrepare:
- push_destination: true
  ...
  includes:
  - nova-compute
  modify_role: tripleo-modify-image
  modify_append_tag: "-hotfix"
  modify_vars:
    tasks_from: rpm_install.yml
    rpms_path: /home/stack/nova-hotfix-pkgs
  ...
```

7.9. MODIFYING CONTAINER IMAGES WITH A CUSTOM DOCKERFILE

For maximum flexibility, you can specify a directory containing a Dockerfile to make the required changes. When you invoke the **tripleo-modify-image** role, the role generates a **Dockerfile.modified** file that changes the **FROM** directive and adds extra **LABEL** directives. The following example runs the custom Dockerfile on the **nova-compute** image:

```
ContainerImagePrepare:
- push_destination: true
  ...
  includes:
  - nova-compute
  modify_role: tripleo-modify-image
  modify_append_tag: "-hotfix"
  modify_vars:
    tasks_from: modify_image.yml
    modify_dir_path: /home/stack/nova-custom
  ...
```

An example **/home/stack/nova-custom/Dockerfile** follows. After running any **USER** root directives, you must switch back to the original image default user:

```
FROM registry.access.redhat.com/rhosp15/openstack-nova-compute:latest
```

```
USER "root"

COPY customize.sh /tmp/
RUN /tmp/customize.sh

USER "nova"
```

CHAPTER 8. BASIC NETWORK ISOLATION

This chapter shows you how to configure the overcloud with the standard network isolation configuration. This includes the following configurations:

- The environment file to enable network isolation (**/usr/share/openstack-tripleo-heat-templates/environments/network-isolation.yaml**).
- The environment file to configure network defaults (**/usr/share/openstack-tripleo-heat-templates/environments/network-environment.yaml**).
- A **network_data** file to define network settings such as IP ranges, subnets, and virtual IPs. This example shows you how to create a copy of the default and edit it to suit your own network.
- Templates to define your NIC layout for each node. The overcloud core template collection contains a set of defaults for different use cases.
- An environment file to enable NICs. This example uses a default file located in the **environments** directory.
- Any additional environment files to customize your networking parameters.

The following content in this chapter shows how to define each of these aspects.

8.1. NETWORK ISOLATION

The overcloud assigns services to the provisioning network by default. However, the director can divide overcloud network traffic into isolated networks. To use isolated networks, the overcloud contains an environment file that enables this feature. The **environments/network-isolation.j2.yaml** file in the director's core Heat templates is a Jinja2 file that defines all ports and VIPs for each network in your composable network file. When rendered, it results in a **network-isolation.yaml** file in the same location with the full resource registry. For example:

```
resource_registry:
  # networks as defined in network_data.yaml
  OS::TripleO::Network::Storage: ../network/storage.yaml
  OS::TripleO::Network::StorageMgmt: ../network/storage_mgmt.yaml
  OS::TripleO::Network::InternalApi: ../network/internal_api.yaml
  OS::TripleO::Network::Tenant: ../network/tenant.yaml
  OS::TripleO::Network::External: ../network/external.yaml

  # Port assignments for the VIPs
  OS::TripleO::Network::Ports::StorageVipPort: ../network/ports/storage.yaml
  OS::TripleO::Network::Ports::StorageMgmtVipPort: ../network/ports/storage_mgmt.yaml
  OS::TripleO::Network::Ports::InternalApiVipPort: ../network/ports/internal_api.yaml
  OS::TripleO::Network::Ports::ExternalVipPort: ../network/ports/external.yaml
  OS::TripleO::Network::Ports::RedisVipPort: ../network/ports/vip.yaml

  # Port assignments by role, edit role definition to assign networks to roles.
  # Port assignments for the Controller
  OS::TripleO::Controller::Ports::StoragePort: ../network/ports/storage.yaml
  OS::TripleO::Controller::Ports::StorageMgmtPort: ../network/ports/storage_mgmt.yaml
  OS::TripleO::Controller::Ports::InternalApiPort: ../network/ports/internal_api.yaml
  OS::TripleO::Controller::Ports::TenantPort: ../network/ports/tenant.yaml
  OS::TripleO::Controller::Ports::ExternalPort: ../network/ports/external.yaml
```

```
# Port assignments for the Compute
OS::TripleO::Compute::Ports::StoragePort: ../network/ports/storage.yaml
OS::TripleO::Compute::Ports::InternalApiPort: ../network/ports/internal_api.yaml
OS::TripleO::Compute::Ports::TenantPort: ../network/ports/tenant.yaml

# Port assignments for the CephStorage
OS::TripleO::CephStorage::Ports::StoragePort: ../network/ports/storage.yaml
OS::TripleO::CephStorage::Ports::StorageMgmtPort: ../network/ports/storage_mgmt.yaml
```

The first section of this file has the resource registry declaration for the **OS::TripleO::Network::*** resources. By default, these resources use the **OS::Heat::None** resource type, which does not create any networks. By redirecting these resources to the YAML files for each network, you enable the creation of these networks.

The next several sections create the IP addresses for the nodes in each role. The controller nodes have IPs on each network. The compute and storage nodes each have IPs on a subset of the networks.

Other functions of overcloud networking, such as Chapter 9, *Custom composable networks* and Chapter 10, *Custom network interface templates* rely on this network isolation environment file. As a result, you need to include the name of the rendered file with your deployment commands. For example:

```
$ openstack overcloud deploy --templates \
  ...
  -e /usr/share/openstack-tripleo-heat-templates/environments/network-isolation.yaml \
  ...
```

8.2. MODIFYING ISOLATED NETWORK CONFIGURATION

The **network_data** file provides a method to configure the default isolated networks. This procedure shows how to create a custom **network_data** file and configure it according to your network requirements.

Procedure

1. Copy the default **network_data** file:

   ```
   $ cp /usr/share/openstack-tripleo-heat-templates/network_data.yaml /home/stack/.
   ```

2. Edit the local copy of the **network_data.yaml** file and modify the parameters to suit your networking requirements. For example, the Internal API network contains the following default network details:

   ```
   - name: InternalApi
     name_lower: internal_api
     vip: true
     vlan: 201
     ip_subnet: '172.16.2.0/24'
     allocation_pools: [{'start': '172.16.2.4', 'end': '172.16.2.250'}]
   ```

Edit the following for each network:

- **vlan** defines the VLAN ID to use for this network.

- **ip_subnet** and **ip_allocation_pools** set the default subnet and IP range for the network..
- **gateway** sets the gateway for the network. Used mostly to define the default route for the External network, but can be used for other networks if necessary.

Include the custom **network_data** file with your deployment using the **-n** option. Without the **-n** option, the deployment command uses the default network details.

8.3. NETWORK INTERFACE TEMPLATES

The overcloud network configuration requires a set of the network interface templates. These templates are standard Heat templates in YAML format. Each role requires a NIC template so the director can configure each node within that role correctly.

All NIC templates contain the same sections as standard Heat templates:

heat_template_version

The syntax version to use.

description

A string description of the template.

parameters

Network parameters to include in the template.

resources

Takes parameters defined in **parameters** and applies them to a network configuration script.

outputs

Renders the final script used for configuration.

The default NIC templates in **/usr/share/openstack-tripleo-heat-templates/networking/config** take advantage of Jinja2 syntax to help render the template. For example, the following snippet from the **single-nic-vlans** configuration renders a set of VLANs for each network:

```
{%- for network in networks if network.enabled|default(true) and network.name in role.networks %}
- type: vlan
  vlan_id:
    get_param: {{network.name}}NetworkVlanID
  addresses:
  - ip_netmask:
      get_param: {{network.name}}IpSubnet
{%- if network.name in role.default_route_networks %}
```

For default Compute nodes, this only renders network information for the Storage, Internal API, and Tenant networks:

```
- type: vlan
  vlan_id:
    get_param: StorageNetworkVlanID
  device: bridge_name
  addresses:
  - ip_netmask:
      get_param: StorageIpSubnet
- type: vlan
  vlan_id:
```

```
      get_param: InternalApiNetworkVlanID
    device: bridge_name
    addresses:
    - ip_netmask:
        get_param: InternalApiIpSubnet
  - type: vlan
    vlan_id:
      get_param: TenantNetworkVlanID
    device: bridge_name
    addresses:
    - ip_netmask:
        get_param: TenantIpSubnet
```

Chapter 10, *Custom network interface templates* explores how to render the default Jinja2-based templates to standard YAML versions, which you can use as a basis for customization.

8.4. DEFAULT NETWORK INTERFACE TEMPLATES

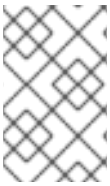
The director contains templates in **/usr/share/openstack-tripleo-heat-templates/network/config/** to suit most common network scenarios. The following table outlines each NIC template set and the respective environment file to use to enable the templates.

NOTE

Each environment file for enabling NIC templates uses the suffix **.j2.yaml**. This is the unrendered Jinja2 version. Ensure that you include the rendered file name, which only uses the **.yaml** suffix, in your deployment.

NIC directory	Description	Environment file
single-nic-vlans	Single NIC (**nic1**) with control plane and VLANs attached to default Open vSwitch bridge.	**environments/net-single-nic-with-vlans.j2.yaml**
single-nic-linux-bridge-vlans	Single NIC (**nic1**) with control plane and VLANs attached to default Linux bridge.	**environments/net-single-nic-linux-bridge-with-vlans**
bond-with-vlans	Control plane attached to **nic1**. Default Open vSwitch bridge with bonded NIC configuration (**nic2** and **nic3**) and VLANs attached.	**environments/net-bond-with-vlans.yaml**

NIC directory	Description	Environment file
multiple-nics	Control plane attached to **nic1**. Assigns each sequential NIC to each network defined in the **network_data** file. By default, this is Storage to **nic2**, Storage Management to **nic3**, Internal API to **nic4**, Tenant to **nic5** on the **br-tenant** bridge, and External to **nic6** on the default Open vSwitch bridge.	**environments/net-multiple-nics.yaml**



NOTE

Environment files exist for using no external network, for example, **net-bond-with-vlans-no-external.yaml**, and using IPv6, for example, **net-bond-with-vlans-v6.yaml**. These are provided for backwards compatibility and do not function with composable networks.

Each default NIC template set contains a **role.role.j2.yaml** template. This file uses Jinja2 to render additional files for each composable role. For example, if your overcloud uses Compute, Controller, and Ceph Storage roles, the deployment renders new templates based on **role.role.j2.yaml**, such as the following templates:

- **compute.yaml**
- **controller.yaml**
- **ceph-storage.yaml**.

8.5. ENABLING BASIC NETWORK ISOLATION

This procedure shows you how to enable basic network isolation using one of the default NIC templates. In this case, it is the single NIC with VLANs template (**single-nic-vlans**).

Procedure

1. When running the **openstack overcloud deploy** command, ensure that you include the rendered environment file names for the following files:
   - The custom **network_data** file.
   - The rendered file name of the default network isolation.
   - The rendered file name of the default network environment file.
   - The rendered file name of the default network interface configuration
   - Any additional environment files relevant to your configuration.

For example:

```
$ openstack overcloud deploy --templates \
```

```
...
-n /home/stack/network_data.yaml \
-e /usr/share/openstack-tripleo-heat-templates/environments/network-isolation.yaml \
-e /usr/share/openstack-tripleo-heat-templates/environments/network-environment.yaml \
-e /usr/share/openstack-tripleo-heat-templates/environments/net-single-nic-with-vlans.yaml \
...
```

CHAPTER 9. CUSTOM COMPOSABLE NETWORKS

This chapter follows on from the concepts and procedures outlined in Chapter 8, *Basic network isolation* and shows you how to configure the overcloud with an additional composable network. This includes configuration of the following files and templates:

- The environment file to enable network isolation (**/usr/share/openstack-tripleo-heat-templates/environments/network-isolation.yaml**).
- The environment file to configure network defaults (**/usr/share/openstack-tripleo-heat-templates/environments/network-environment.yaml**).
- A custom **network_data** file to create additional networks outside of the defaults.
- A custom **roles_data** file to assign custom networks to roles.
- Templates to define your NIC layout for each node. The overcloud core template collection contains a set of defaults for different use cases.
- An environment file to enable NICs. This example uses a a default file located in the **environments** directory.
- Any additional environment files to customize your networking parameters. This example uses an environment file to customize OpenStack service mappings to composable networks.

The following content in this chapter shows you how to define each of these aspects.

9.1. COMPOSABLE NETWORKS

The overcloud uses the following pre-defined set of network segments by default:

- Control Plane
- Internal API
- Storage
- Storage Management
- Tenant
- External
- Management (optional)

You can use Composable networks to add networks for various services. For example, if you have a network dedicated to NFS traffic, you can present it to multiple roles.

Director supports the creation of custom networks during the deployment and update phases. These additional networks can be used for ironic bare metal nodes, system management, or to create separate networks for different roles. You can also use them to create multiple sets of networks for split deployments where traffic is routed between networks.

A single data file (**network_data.yaml**) manages the list of networks to be deployed. Include this file with your deployment command using the **-n** option. Without this option, the deployment uses the default file (**/usr/share/openstack-tripleo-heat-templates/network_data.yaml**).

9.2. ADDING A COMPOSABLE NETWORK

This procedure shows you how to add an additional composable network to your overcloud.

Procedure

1. Copy the default **network_data** file:

```
$ cp /usr/share/openstack-tripleo-heat-templates/network_data.yaml /home/stack/.
```

2. Edit the local copy of the **network_data.yaml** file and add a section for your new network. For example:

```
- name: StorageBackup
  vip: true
  name_lower: storage_backup
  ip_subnet: '172.21.1.0/24'
  allocation_pools: [{'start': '171.21.1.4', 'end': '172.21.1.250'}]
  gateway_ip: '172.21.1.1'
```

   - **name** is the only mandatory value, however you can also use **name_lower** to normalize names for readability. For example, changing **InternalApi** to **internal_api**.
   - **vip: true** creates a virtual IP address (VIP) on the new network. This IP is used as the target IP for services listed in the service-to-network mapping parameter (**ServiceNetMap**). Note that VIPs are only used by roles that use Pacemaker. The overcloud's load-balancing service redirects traffic from these IPs to their respective service endpoint.
   - **ip_subnet**, **allocation_pools**, and **gateway_ip** set the default IPv4 subnet, IP range, and gateway for the network.

Include the custom **network_data** file with your deployment using the **-n** option. Without the **-n** option, the deployment command uses the default set of networks.

9.3. INCLUDING A COMPOSABLE NETWORK IN A ROLE

You can assign composable networks to the roles defined in your environment. For example, you might include a custom **StorageBackup** network with your Ceph Storage nodes.

This procedure shows you how to add composable networks to a role in your overcloud.

Procedure

1. If you do not already have a custom **roles_data** file, copy the default to your home directory:

```
$ cp /usr/share/openstack-tripleo-heat-templates/roles_data.yaml /home/stack/.
```

2. Edit the custom **roles_data** file.
3. Scroll to the role you want to add the composable network and add the network name to the list of **networks**. For example, to add the network to the Ceph Storage role, use the following snippet as a guide:

```
- name: CephStorage
```

```
description: |
  Ceph OSD Storage node role
networks:
  - Storage
  - StorageMgmt
  - StorageBackup
```

4. After adding custom networks to their respective roles, save the file.

When running the **openstack overcloud deploy** command, include the **roles_data** file using the **-r** option. Without the **-r** option, the deployment command uses the default set of roles with their respective assigned networks.

9.4. ASSIGNING OPENSTACK SERVICES TO COMPOSABLE NETWORKS

Each OpenStack service is assigned to a default network type in the resource registry. These services are then bound to IP addresses within the network type's assigned network. Although the OpenStack services are divided among these networks, the number of actual physical networks can differ as defined in the network environment file. You can reassign OpenStack services to different network types by defining a new network map in an environment file, for example, **/home/stack/templates/service-reassignments.yaml**. The **ServiceNetMap** parameter determines the network types used for each service.

For example, you can reassign the Storage Management network services to the Storage Backup Network by modifying the highlighted sections:

```
parameter_defaults:
  ServiceNetMap:
    SwiftMgmtNetwork: storage_backup
    CephClusterNetwork: storage_backup
```

Changing these parameters to **storage_backup** places these services on the Storage Backup network instead of the Storage Management network. This means you only need to define a set of **parameter_defaults** for the Storage Backup network and not the Storage Management network.

The director merges your custom **ServiceNetMap** parameter definitions into a pre-defined list of defaults taken from **ServiceNetMapDefaults** and overrides the defaults. The director returns the full list, including customizations back to **ServiceNetMap**, which is used to configure network assignments for various services.

Service mappings apply to networks that use **vip: true** in the **network_data** file for nodes that use Pacemaker. The overcloud's load balancer redirects traffic from the VIPs to the specific service endpoints.

NOTE

A full list of default services can be found in the **ServiceNetMapDefaults** parameter within **/usr/share/openstack-tripleo-heat-templates/network/service_net_map.j2.yaml**.

9.5. ENABLING CUSTOM COMPOSABLE NETWORKS

This procedure shows you how to enable custom composable networks using one of the default NIC templates. In this case, it is the Single NIC with VLANs (**single-nic-vlans**).

Procedure

1. When you run the **openstack overcloud deploy** command, ensure that you include the following files:
   - The custom **network_data** file.
   - The custom **roles_data** file with network-to-role assignments.
   - The rendered file name of the default network isolation.
   - The rendered file name of the default network environment file.
   - The rendered file name of the default network interface configuration.
   - Any additional environment files related to your network, such as the service reassignments.

For example:

```
$ openstack overcloud deploy --templates \
  ...
  -n /home/stack/network_data.yaml \
  -r /home/stack/roles_data.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/network-isolation.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/network-environment.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/net-single-nic-with-vlans.yaml \
  -e /home/stack/templates/service-reassignments.yaml \
  ...
```

This example command deploys the composable networks, including your additional custom networks, across nodes in your overcloud.

CHAPTER 10. CUSTOM NETWORK INTERFACE TEMPLATES

This chapter follows on from the concepts and procedures outlined in Chapter 8, *Basic network isolation*. The purpose of this chapter is to demonstrate how to create a set of custom network interface template to suit nodes in your environment. This includes:

- The environment file to enable network isolation (**/usr/share/openstack-tripleo-heat-templates/environments/network-isolation.yaml**).
- The environment file to configure network defaults (**/usr/share/openstack-tripleo-heat-templates/environments/network-environment.yaml**).
- Templates to define your NIC layout for each node. The overcloud core template collection contains a set of defaults for different use cases. In this situation, you will render a default a basis for your custom templates.
- A custom environment file to enable NICs. This example uses a custom environment file (**/home/stack/templates/custom-network-configuration.yaml**) that references your custom interface templates.
- Any additional environment files to customize your networking parameters.
- If using customizing your networks, a custom **network_data** file.
- If creating additional or custom composable networks, a custom **network_data** file and a custom **roles_data** file.

10.1. CUSTOM NETWORK ARCHITECTURE

The default NIC templates might not suit a specific network configuration. For example, you might want to create your own custom NIC template that suits a specific network layout. You might aim to separate the control services and data services on to separate NICs. In this situation, the service to NIC assignments result in the following mapping:

- NIC1 (Provisioning):
  - Provisioning / Control Plane
- NIC2 (Control Group)
  - Internal API
  - Storage Management
  - External (Public API)
- NIC3 (Data Group)
  - Tenant Network (VXLAN tunneling)
  - Tenant VLANs / Provider VLANs
  - Storage
  - External VLANs (Floating IP/SNAT)
- NIC4 (Management)

- Management

10.2. RENDERING DEFAULT NETWORK INTERFACE TEMPLATES FOR CUSTOMIZATION

For the purposes of simplifying the configuration of custom interface templates, this procedure shows you how to render the Jinja2 syntax of a default NIC template. This way you can use the rendered templates as a basis for your custom configuration.

Procedure

1. Render a copy of the **openstack-tripleo-heat-templates** collection using the **process-templates.py** script:

```
$ cd /usr/share/openstack-tripleo-heat-templates
$ ./tools/process-templates.py -o ~/openstack-tripleo-heat-templates-rendered
```

   This converts all Jinja2 templates to their rendered YAML versions and saves the results to **~/openstack-tripleo-heat-templates-rendered**.

   If using a custom network file or custom roles file, you can include these files using the **-n** and **-r** options respectively. For example:

```
$ ./tools/process-templates.py -o ~/openstack-tripleo-heat-templates-rendered -n
/home/stack/network_data.yaml -r /home/stack/roles_data.yaml
```

2. Copy the multiple NIC example:

```
$ cp -r ~/openstack-tripleo-heat-templates-rendered/network/config/multiple-nics/
~/templates/custom-nics/
```

3. You can edit the template set in **custom-nics** to suit your own network configuration.

10.3. NETWORK INTERFACE ARCHITECTURE

This section explores the architecture of the custom NIC templates in **custom-nics** and provides recommendations on editing them.

Parameters

The **parameters** section contains all network configuration parameters for network interfaces. This includes information such as subnet ranges and VLAN IDs. This section should remain unchanged as the Heat template inherits values from its parent template. However, you can modify the values for some parameters using a network environment file.

Resources

The **resources** section is where the main network interface configuration occurs. In most cases, the **resources** section is the only one that requires editing. Each **resources** section begins with the following header:

```
resources:
  OsNetConfigImpl:
    type: OS::Heat::SoftwareConfig
```

```
properties:
  group: script
  config:
    str_replace:
      template:
        get_file: /usr/share/openstack-tripleo-heat-templates/network/scripts/run-os-net-config.sh
      params:
        $network_config:
          network_config:
```

This runs a script (**run-os-net-config.sh**) that creates a configuration file for **os-net-config** to use for configuring network properties on a node. The **network_config** section contains the custom network interface data sent to the **run-os-net-config.sh** script. You arrange this custom interface data in a sequence based on the type of device.

IMPORTANT

If creating custom NIC templates, you must set the **run-os-net-config.sh** script location to an absolute location for each NIC template. The script is located at **/usr/share/openstack-tripleo-heat-templates/network/scripts/run-os-net-config.sh** on the undercloud.

10.4. NETWORK INTERFACE REFERENCE

The following sections define the network interface types and the parameters used in each.

interface

Defines a single network interface. The configuration defines each interface using either the actual interface name ("eth0", "eth1", "enp0s25") or a set of numbered interfaces ("nic1", "nic2", "nic3").

For example:

```
- type: interface
  name: nic2
```

Table 10.1. interface options

Option	Default	Description
name		Name of the Interface
use_dhcp	False	Use DHCP to get an IP address
use_dhcpv6	False	Use DHCP to get a v6 IP address
addresses		A list of IP addresses assigned to the interface
routes		A list of routes assigned to the interface. See routes.

Option	Default	Description
mtu	1500	The maximum transmission unit (MTU) of the connection
primary	False	Defines the interface as the primary interface
defroute	True	Use a default route provided by the DHCP service. Only applies when **use_dhcp** or **use_dhcpv6** is enabled.
persist_mapping	False	Write the device alias configuration instead of the system names
dhclient_args	None	Arguments to pass to the DHCP client
dns_servers	None	List of DNS servers to use for the interface

vlan

Defines a VLAN. Use the VLAN ID and subnet passed from the **parameters** section.

For example:

```
- type: vlan
  vlan_id:{get_param: ExternalNetworkVlanID}
  addresses:
    - ip_netmask: {get_param: ExternalIpSubnet}
```

Table 10.2. vlan options

Option	Default	Description
vlan_id		The VLAN ID
device		The parent device to attach the VLAN. Use this parameter when the VLAN is not a member of an OVS bridge. For example, use this parameter to attach the VLAN to a bonded interface device.
use_dhcp	False	Use DHCP to get an IP address

Option	Default	Description
use_dhcpv6	False	Use DHCP to get a v6 IP address
addresses		A list of IP addresses assigned to the VLAN
routes		A list of routes assigned to the VLAN. See routes.
mtu	1500	The maximum transmission unit (MTU) of the connection
primary	False	Defines the VLAN as the primary interface
defroute	True	Use a default route provided by the DHCP service. Only applies when **use_dhcp** or **use_dhcpv6** is enabled.
persist_mapping	False	Write the device alias configuration instead of the system names
dhclient_args	None	Arguments to pass to the DHCP client
dns_servers	None	List of DNS servers to use for the VLAN

ovs_bond

Defines a bond in Open vSwitch to join two or more **interfaces** together. This helps with redundancy and increases bandwidth.

For example:

```
- type: ovs_bond
  name: bond1
  members:
  - type: interface
    name: nic2
  - type: interface
    name: nic3
```

Table 10.3. ovs_bond options

Option	Default	Description
name		Name of the bond
use_dhcp	False	Use DHCP to get an IP address
use_dhcpv6	False	Use DHCP to get a v6 IP address
addresses		A list of IP addresses assigned to the bond
routes		A list of routes assigned to the bond. See routes.
mtu	1500	The maximum transmission unit (MTU) of the connection
primary	False	Defines the interface as the primary interface
members		A sequence of interface objects to use in the bond
ovs_options		A set of options to pass to OVS when creating the bond
ovs_extra		A set of options to to set as the OVS_EXTRA parameter in the bond's network configuration file
defroute	True	Use a default route provided by the DHCP service. Only applies when **use_dhcp** or **use_dhcpv6** is enabled.
persist_mapping	False	Write the device alias configuration instead of the system names
dhclient_args	None	Arguments to pass to the DHCP client
dns_servers	None	List of DNS servers to use for the bond

ovs_bridge

Defines a bridge in Open vSwitch, which connects multiple **interface**, **ovs_bond**, and **vlan** objects together. The external bridge also uses two special values for parameters:

- **bridge_name**, which is replaced with the external bridge name.
- **interface_name**, which is replaced with the external interface.

For example:

```
- type: ovs_bridge
  name: bridge_name
  addresses:
  - ip_netmask:
      list_join:
      - /
      - - {get_param: ControlPlaneIp}
        - {get_param: ControlPlaneSubnetCidr}
  members:
    - type: interface
      name: interface_name
- type: vlan
  device: bridge_name
  vlan_id:
    {get_param: ExternalNetworkVlanID}
  addresses:
    - ip_netmask:
        {get_param: ExternalIpSubnet}
```

NOTE

The OVS bridge connects to the Neutron server in order to get configuration data. If the OpenStack control traffic (typically the Control Plane and Internal API networks) is placed on an OVS bridge, then connectivity to the Neutron server gets lost whenever OVS is upgraded or the OVS bridge is restarted by the admin user or process. This will cause some downtime. If downtime is not acceptable under these circumstances, then the Control group networks should be placed on a separate interface or bond rather than on an OVS bridge:

- A minimal setting can be achieved, when you put the Internal API network on a VLAN on the provisioning interface and the OVS bridge on a second interface.
- If you want bonding, you need at least two bonds (four network interfaces). The control group should be placed on a Linux bond (Linux bridge). If the switch does not support LACP fallback to a single interface for PXE boot, then this solution requires at least five NICs.

Table 10.4. ovs_bridge options

Option	Default	Description
name		Name of the bridge
use_dhcp	False	Use DHCP to get an IP address
use_dhcpv6	False	Use DHCP to get a v6 IP address

Option	Default	Description
addresses		A list of IP addresses assigned to the bridge
routes		A list of routes assigned to the bridge. See routes.
mtu	1500	The maximum transmission unit (MTU) of the connection
members		A sequence of interface, VLAN, and bond objects to use in the bridge
ovs_options		A set of options to pass to OVS when creating the bridge
ovs_extra		A set of options to to set as the OVS_EXTRA parameter in the bridge's network configuration file
defroute	True	Use a default route provided by the DHCP service. Only applies when **use_dhcp** or **use_dhcpv6** is enabled.
persist_mapping	False	Write the device alias configuration instead of the system names
dhclient_args	None	Arguments to pass to the DHCP client
dns_servers	None	List of DNS servers to use for the bridge

linux_bond

Defines a Linux bond that joins two or more **interfaces** together. This helps with redundancy and increases bandwidth. Make sure to include the kernel-based bonding options in the **bonding_options** parameter.

For example:

```
- type: linux_bond
  name: bond1
  members:
  - type: interface
```

```
      name: nic2
      primary: true
    - type: interface
      name: nic3
    bonding_options: "mode=802.3ad"
```

Note that **nic2** uses **primary: true**. This ensures the bond uses the MAC address for **nic2**.

Table 10.5. linux_bond options

Option	Default	Description
name		Name of the bond
use_dhcp	False	Use DHCP to get an IP address
use_dhcpv6	False	Use DHCP to get a v6 IP address
addresses		A list of IP addresses assigned to the bond
routes		A list of routes assigned to the bond. See routes.
mtu	1500	The maximum transmission unit (MTU) of the connection
primary	False	Defines the interface as the primary interface.
members		A sequence of interface objects to use in the bond
bonding_options		A set of options when creating the bond.
defroute	True	Use a default route provided by the DHCP service. Only applies when **use_dhcp** or **use_dhcpv6** is enabled.
persist_mapping	False	Write the device alias configuration instead of the system names
dhclient_args	None	Arguments to pass to the DHCP client
dns_servers	None	List of DNS servers to use for the bond

linux_bridge

Defines a Linux bridge, which connects multiple **interface**, **linux_bond**, and **vlan** objects together. The external bridge also uses two special values for parameters:

- **bridge_name**, which is replaced with the external bridge name.
- **interface_name**, which is replaced with the external interface.

For example:

```
- type: linux_bridge
  name: bridge_name
  addresses:
    - ip_netmask:
        list_join:
          - /
          - - {get_param: ControlPlaneIp}
            - {get_param: ControlPlaneSubnetCidr}
  members:
    - type: interface
      name: interface_name
- type: vlan
  device: bridge_name
  vlan_id:
    {get_param: ExternalNetworkVlanID}
  addresses:
    - ip_netmask:
        {get_param: ExternalIpSubnet}
```

Table 10.6. linux_bridge options

Option	Default	Description
name		Name of the bridge
use_dhcp	False	Use DHCP to get an IP address
use_dhcpv6	False	Use DHCP to get a v6 IP address
addresses		A list of IP addresses assigned to the bridge
routes		A list of routes assigned to the bridge. See routes.
mtu	1500	The maximum transmission unit (MTU) of the connection
members		A sequence of interface, VLAN, and bond objects to use in the bridge

Option	Default	Description
defroute	True	Use a default route provided by the DHCP service. Only applies when **use_dhcp** or **use_dhcpv6** is enabled.
persist_mapping	False	Write the device alias configuration instead of the system names
dhclient_args	None	Arguments to pass to the DHCP client
dns_servers	None	List of DNS servers to use for the bridge

routes

Defines a list of routes to apply to a network interface, VLAN, bridge, or bond.

For example:

```
- type: interface
  name: nic2
  ...
  routes:
    - ip_netmask: 10.1.2.0/24
      default: true
      next_hop:
        get_param: EC2MetadataIp
```

Option	Default	Description
ip_netmask	None	IP and netmask of the destination network.
default	False	Sets this this route to a default route. Equivalent to setting **ip_netmask: 0.0.0.0/0**.
next_hop	None	The IP address of the router used to reach the destination network.

10.5. EXAMPLE NETWORK INTERFACE LAYOUT

The following snippet for a possible Controller node NIC template demonstrates how to configure the custom network scenario to keep the control group apart from the OVS bridge:

```
resources:
  OsNetConfigImpl:
    type: OS::Heat::SoftwareConfig
    properties:
      group: script
      config:
        str_replace:
          template:
            get_file: /usr/share/openstack-tripleo-heat-templates/network/scripts/run-os-net-config.sh
          params:
            $network_config:
              network_config:

              # NIC 1 - Provisioning
              - type: interface
                name: nic1
                use_dhcp: false
                addresses:
                - ip_netmask:
                    list_join:
                    - /
                    - - get_param: ControlPlaneIp
                      - get_param: ControlPlaneSubnetCidr
                routes:
                - ip_netmask: 169.254.169.254/32
                  next_hop:
                    get_param: EC2MetadataIp

              # NIC 2 - Control Group
              - type: interface
                name: nic2
                use_dhcp: false
              - type: vlan
                device: nic2
                vlan_id:
                  get_param: InternalApiNetworkVlanID
                addresses:
                - ip_netmask:
                    get_param: InternalApiIpSubnet
              - type: vlan
                device: nic2
                vlan_id:
                  get_param: StorageMgmtNetworkVlanID
                addresses:
                - ip_netmask:
                    get_param: StorageMgmtIpSubnet
              - type: vlan
                device: nic2
                vlan_id:
                  get_param: ExternalNetworkVlanID
                addresses:
                - ip_netmask:
                    get_param: ExternalIpSubnet
                routes:
                - default: true
                  next_hop:
```

```
            get_param: ExternalInterfaceDefaultRoute

        # NIC 3 - Data Group
        - type: ovs_bridge
          name: bridge_name
          dns_servers:
            get_param: DnsServers
          members:
          - type: interface
            name: nic3
            primary: true
          - type: vlan
            vlan_id:
              get_param: StorageNetworkVlanID
            addresses:
            - ip_netmask:
                get_param: StorageIpSubnet
          - type: vlan
            vlan_id:
              get_param: TenantNetworkVlanID
            addresses:
            - ip_netmask:
                get_param: TenantIpSubnet

          # NIC 4 - Management
          - type: interface
            name: nic4
            use_dhcp: false
            addresses:
            - ip_netmask: {get_param: ManagementIpSubnet}
            routes:
            - default: true
              next_hop: {get_param: ManagementInterfaceDefaultRoute}
```

This template uses four network interfaces and assigns a number of tagged VLAN devices to the numbered interfaces, **nic1** to **nic4**. On **nic3** it creates the OVS bridge that hosts the Storage and Tenant networks. As a result, it creates the following layout:

- NIC1 (Provisioning):
  - Provisioning / Control Plane
- NIC2 (Control Group)
  - Internal API
  - Storage Management
  - External (Public API)
- NIC3 (Data Group)
  - Tenant Network (VXLAN tunneling)
  - Tenant VLANs / Provider VLANs
  - Storage

    - External VLANs (Floating IP/SNAT)
- NIC4 (Management)
    - Management

10.6. NETWORK INTERFACE TEMPLATE CONSIDERATIONS FOR CUSTOM NETWORKS

When using composable networks, the **process-templates.py** script renders the static templates to include networks and roles defined in your **network_data** and **roles_data** files. Check the rendered NIC templates and ensure it contains:

- Static file for each roles, including custom composable networks.
- Each static file for each role contains the correct network definitions.

Each static file requires all the parameter definitions for any custom networks even if the network is not used on the role. Check to make sure the rendered templates contain these parameters. For example, if a **StorageBackup** network is added to only the Ceph nodes, the **parameters** section in NIC configuration templates for all roles must also include this definition:

```
parameters:
  ...
  StorageBackupIpSubnet:
    default: ''
    description: IP address/subnet on the external network
    type: string
  ...
```

You can also include the **parameters** definitions for VLAN IDs and/or gateway IP, if needed:

```
parameters:
  ...
  StorageBackupNetworkVlanID:
    default: 60
    description: Vlan ID for the management network traffic.
    type: number
  StorageBackupDefaultRoute:
   description: The default route of the storage backup network.
   type: string
  ...
```

The **IpSubnet** parameter for the custom network appears in the parameter definitions for each role. However, since the Ceph role might be the only role that uses the **StorageBackup** network, only the NIC configuration template for the Ceph role would make use of the **StorageBackup** parameters in the **network_config** section of the template.

```
    $network_config:
      network_config:
      - type: interface
        name: nic1
        use_dhcp: false
```

```
    addresses:
    - ip_netmask:
        get_param: StorageBackupIpSubnet
```

10.7. CUSTOM NETWORK ENVIRONMENT FILE

The custom network environment file (in this case, **/home/stack/templates/custom-network-configuration.yaml**) is a Heat environment file that describes the Overcloud's network environment and points to the custom network interface configuration templates. You can define the subnets and VLANs for your network along with IP address ranges. You can then customize these values for the local environment.

The **resource_registry** section contains references to the custom network interface templates for each node role. Each resource registered uses the following format:

- **OS::TripleO::[ROLE]::Net::SoftwareConfig: [FILE]**

[ROLE] is the role name and **[FILE]** is the respective network interface template for that particular role. For example:

```
resource_registry:
  OS::TripleO::Controller::Net::SoftwareConfig: /home/stack/templates/custom-nics/controller.yaml
```

The **parameter_defaults** section contains a list of parameters that define the network options for each network type.

10.8. NETWORK ENVIRONMENT PARAMETERS

The following table is a list of parameters you can use in a network environment file's **parameter_defaults** section to override the default parameter values in your NIC templates.

Parameter	Description	Type
ControlPlaneDefaultRoute	The IP address of the router on the Control Plane, which is used as a default route for roles other than the Controller nodes by default. Set to the undercloud IP if using IP masquerade instead of a router.	string
ControlPlaneSubnetCidr	The CIDR netmask of the IP network used on the Control Plane. If the Control Plane network uses 192.168.24.0/24, the CIDR is **24**.	string (though is always a number)

Parameter	Description	Type
***NetCidr**	The full network and CIDR netmask for a particular network. The default is automatically set to the network's **ip_subnet** setting in the **network_data** file. For example: **InternalApiNetCidr: 172.16.0.0/24**	string
***AllocationPools**	"The IP allocation range for a particular network. The default is automatically set to the network's **allocation_pools** setting in the **network_data** file. For example: **InternalApiAllocationPools: [{'start': '172.16.0.10', 'end': '172.16.0.200'}]**	hash
***NetworkVlanID**	The node's VLAN ID for on a particular network. The default is set automatically to the network's **vlan** setting in the **network_data** file. For example: **InternalApiNetworkVlanID: 201**	number
***InterfaceDefaultRoute**	The router address for a particular network, which you can use as a default route for roles or used for routes to other networks. The default is automatically set to the network's **gateway_ip** setting in the **network_data** file. For example: **InternalApiInterfaceDefaultRoute: 172.16.0.1**	string
DnsServers	A list of DNS servers added to resolv.conf. Usually allows a maximum of 2 servers.	comma delimited list
EC2MetadataIp	The IP address of the metadata server used to provision overcloud nodes. Set to the IP address of the undercloud on the Control Plane.	string

Parameter	Description	Type
BondInterfaceOvsOptions	The options for bonding interfaces. For example: **BondInterfaceOvsOptions: "bond_mode=balance-slb"**	string
NeutronExternalNetworkBridge	Legacy value for the name of the external bridge to use for OpeNStack Networking (neutron). This value is empty by default, which allows for multiple physical bridges to be defined in the **NeutronBridgeMappings**. This should not normally be overridden.	string
NeutronFlatNetworks	Defines the flat networks to configure in neutron plugins. Defaults to "datacentre" to permit external network creation. For example: **NeutronFlatNetworks: "datacentre"**	string
NeutronBridgeMappings	The logical to physical bridge mappings to use. Defaults to mapping the external bridge on hosts (**br-ex**) to a physical name (**datacentre**). You would refer to the logical name when creating OpenStack Networking (neutron) provider networks or floating IP networks. For example **NeutronBridgeMappings: "datacentre:br-ex,tenant:br-tenant"**	string
NeutronPublicInterface	Defines the interface to bridge onto **br-ex** for network nodes when not using network isolation. Usually not used except in small deployments with only two networks. For example: **NeutronPublicInterface: "eth0"**	string

Parameter	Description	Type
NeutronNetworkType	TThe tenant network type for OpenStack Networking (neutron). To specify multiple values, use a comma separated list. The first type specified is used until all available networks are exhausted, then the next type is used. For example: **NeutronNetworkType: "vxlan"**	string
NeutronTunnelTypes	The tunnel types for the neutron tenant network. To specify multiple values, use a comma separated string. For example: *NeutronTunnelTypes: 'gre,vxlan'*	string / comma separated list
NeutronTunnelIdRanges	Ranges of GRE tunnel IDs to make available for tenant network allocation. For example: **NeutronTunnelIdRanges "1:1000"**	string
NeutronVniRanges	Ranges of VXLAN VNI IDs to make available for tenant network allocation. For example: **NeutronVniRanges: "1:1000"**	string
NeutronEnableTunnelling	Defines whether to enable or completely disable all tunnelled networks. Leave this enabled unless you are sure you will never want to create tunelled networks. Defaults to enabled.	Boolean
NeutronNetworkVLANRanges	The ML2 and Open vSwitch VLAN mapping range to support. Defaults to permitting any VLAN on the **datacentre** physical network. To specify multiple values, use a comma separated list. For example: **NeutronNetworkVLANRanges: "datacentre:1:1000,tenant:100:299,tenant:310:399"**	string

Parameter	Description	Type
NeutronMechanismDrivers	The mechanism drivers for the neutron tenant network. Defaults to "openvswitch". To specify multiple values, use a comma-separated string. For example: **NeutronMechanismDrivers: 'openvswitch,l2population'**	string / comma separated list

10.9. EXAMPLE CUSTOM NETWORK ENVIRONMENT FILE

The following is an example of an environment file to enable your NIC templates and set custom parameters.

```
resource_registry:
  OS::TripleO::BlockStorage::Net::SoftwareConfig:
    /home/stack/templates/nic-configs/cinder-storage.yaml
  OS::TripleO::Compute::Net::SoftwareConfig:
    /home/stack/templates/nic-configs/compute.yaml
  OS::TripleO::Controller::Net::SoftwareConfig:
    /home/stack/templates/nic-configs/controller.yaml
  OS::TripleO::ObjectStorage::Net::SoftwareConfig:
    /home/stack/templates/nic-configs/swift-storage.yaml
  OS::TripleO::CephStorage::Net::SoftwareConfig:
    /home/stack/templates/nic-configs/ceph-storage.yaml

parameter_defaults:
  # Gateway router for the provisioning network (or Undercloud IP)
  ControlPlaneDefaultRoute: 192.0.2.254
  # The IP address of the EC2 metadata server. Generally the IP of the Undercloud
  EC2MetadataIp: 192.0.2.1
  # Define the DNS servers (maximum 2) for the overcloud nodes
  DnsServers: ["8.8.8.8","8.8.4.4"]
  NeutronExternalNetworkBridge: """
```

10.10. ENABLING NETWORK ISOLATION WITH CUSTOM NICS

This procedure show how to enable network isolation using custom NIC templates.

Procedure

1. When running the **openstack overcloud deploy** command, make sure to include:
   - The custom **network_data** file.
   - The rendered file name of the default network isolation.
   - The rendered file name of the default network environment file.
   - The custom environment network configuration that includes resource references to your custom NIC templates.

- Any additional environment files relevant to your configuration.

For example:

```
$ openstack overcloud deploy --templates \
  ...
  -n /home/stack/network_data.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/network-isolation.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/network-environment.yaml \
  -e /home/stack/templates/custom-network-configuration.yaml \
  ...
```

- Include the **network-isolation.yaml** file first, then the **network-environment.yaml** file. The subsequent **custom-network-configuration.yaml** overrides the **OS::TripleO::[ROLE]::Net::SoftwareConfig** resources from the previous two files..

- If using composable networks, include the **network_data** and **roles_data** files with this command.

CHAPTER 11. ADDITIONAL NETWORK CONFIGURATION

This chapter follows on from the concepts and procedures outlined in Chapter 10, *Custom network interface templates* and provides some additional information to help configure parts of your overcloud network.

11.1. CONFIGURING CUSTOM INTERFACES

Individual interfaces might require modification. The example below shows modifications required to use the second NIC to connect to an infrastructure network with DHCP addresses, and to use the third and fourth NICs for the bond:

```
network_config:
  # Add a DHCP infrastructure network to nic2
  - type: interface
    name: nic2
    use_dhcp: true
  - type: ovs_bridge
    name: br-bond
    members:
      - type: ovs_bond
        name: bond1
        ovs_options:
          get_param: BondInterfaceOvsOptions
        members:
          # Modify bond NICs to use nic3 and nic4
          - type: interface
            name: nic3
            primary: true
          - type: interface
            name: nic4
```

The network interface template uses either the actual interface name (**eth0**, **eth1**, **enp0s25**) or a set of numbered interfaces (**nic1**, **nic2**, **nic3**). The network interfaces of hosts within a role do not have to be exactly the same when using numbered interfaces (**nic1**, **nic2**, etc.) instead of named interfaces (**eth0**, **eno2**, etc.). For example, one host might have interfaces **em1** and **em2**, while another has **eno1** and **eno2**, but you can refer to the NICs of both hosts as **nic1** and **nic2**.

The order of numbered interfaces corresponds to the order of named network interface types:

- **ethX** interfaces, such as **eth0**, **eth1**, etc. These are usually onboard interfaces.
- **enoX** interfaces, such as **eno0**, **eno1**, etc. These are usually onboard interfaces.
- **enX** interfaces, sorted alpha numerically, such as **enp3s0**, **enp3s1**, **ens3**, etc. These are usually add-on interfaces.

The numbered NIC scheme only takes into account the interfaces that are live, for example, if they have a cable attached to the switch. If you have some hosts with four interfaces and some with six interfaces, you should use **nic1** to **nic4** and only plug four cables on each host.

You can hardcode physical interfaces to specific aliases. This allows you to be pre-determine which physical NIC will map as **nic1** or **nic2** and so on. You can also map a MAC address to a specified alias.

NOTE

Normally, **os-net-config** only registers interfaces that are already connected in an **UP** state. However, if you hardcode interfaces using a custom mapping file, the interface is registered even if it is in a **DOWN** state.

Interfaces are mapped to aliases using an environment file. In this example, each node has predefined entries for **nic1** and **nic2**:

```
parameter_defaults:
  NetConfigDataLookup:
    node1:
      nic1: "em1"
      nic2: "em2"
    node2:
      nic1: "00:50:56:2F:9F:2E"
      nic2: "em2"
```

The resulting configuration is applied by **os-net-config**. On each node, you can see the applied configuration under **interface_mapping** in **/etc/os-net-config/mapping.yaml**.

11.2. CONFIGURING ROUTES AND DEFAULT ROUTES

There are two ways to set the default route of a host. If the interface is using DHCP and the DHCP server offers a gateway address, the system uses a default route for that gateway. Otherwise, you can set a default route on an interface with a static IP.

Although the Linux kernel supports multiple default gateways, it only uses the one with the lowest metric. If there are multiple DHCP interfaces, this can result in an unpredictable default gateway. In this case, it is recommended to set **defroute: false** for interfaces other than the one using the default route.

For example, you might want a DHCP interface (**nic3**) to be the default route. Use the following YAML to disable the default route on another DHCP interface (**nic2**):

```
# No default route on this DHCP interface
- type: interface
  name: nic2
  use_dhcp: true
  defroute: false
# Instead use this DHCP interface as the default route
- type: interface
  name: nic3
  use_dhcp: true
```

NOTE

The **defroute** parameter only applies to routes obtained through DHCP.

To set a static route on an interface with a static IP, specify a route to the subnet. For example, you can set a route to the 10.1.2.0/24 subnet through the gateway at 172.17.0.1 on the Internal API network:

```
  - type: vlan
    device: bond1
```

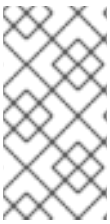
```
    vlan_id:
      get_param: InternalApiNetworkVlanID
    addresses:
    - ip_netmask:
        get_param: InternalApiIpSubnet
    routes:
    - ip_netmask: 10.1.2.0/24
      next_hop: 172.17.0.1
```

11.3. CONFIGURING JUMBO FRAMES

The Maximum Transmission Unit (MTU) setting determines the maximum amount of data transmitted with a single Ethernet frame. Using a larger value results in less overhead since each frame adds data in the form of a header. The default value is 1500 and using a higher value requires the configuration of the switch port to support jumbo frames. Most switches support an MTU of at least 9000, but many are configured for 1500 by default.

The MTU of a VLAN cannot exceed the MTU of the physical interface. Ensure that you include the MTU value on the bond and/or interface.

The Storage, Storage Management, Internal API, and Tenant networks all benefit from jumbo frames. In testing, a project's networking throughput demonstrated substantial improvement when using jumbo frames in conjunction with VXLAN tunnels.



NOTE

It is recommended that the Provisioning interface, External interface, and any floating IP interfaces be left at the default MTU of 1500. Connectivity problems are likely to occur otherwise. This is because routers typically cannot forward jumbo frames across Layer 3 boundaries.

```
- type: ovs_bond
  name: bond1
  mtu: 9000
  ovs_options: {get_param: BondInterfaceOvsOptions}
  members:
    - type: interface
      name: nic3
      mtu: 9000
      primary: true
    - type: interface
      name: nic4
      mtu: 9000

# The external interface should stay at default
- type: vlan
  device: bond1
  vlan_id:
    get_param: ExternalNetworkVlanID
  addresses:
    - ip_netmask:
        get_param: ExternalIpSubnet
  routes:
    - ip_netmask: 0.0.0.0/0
```

```
      next_hop:
        get_param: ExternalInterfaceDefaultRoute

# MTU 9000 for Internal API, Storage, and Storage Management
- type: vlan
  device: bond1
  mtu: 9000
  vlan_id:
    get_param: InternalApiNetworkVlanID
  addresses:
  - ip_netmask:
      get_param: InternalApiIpSubnet
```

11.4. CONFIGURING THE NATIVE VLAN FOR FLOATING IPS

Neutron uses a default empty string for its external bridge mapping. This maps the physical interface to the **br-int** instead of using **br-ex** directly. This model allows multiple Floating IP networks using either VLANs or multiple physical connections.

Use the **NeutronExternalNetworkBridge** parameter in the **parameter_defaults** section of your network isolation environment file:

```
parameter_defaults:
  # Set to "br-ex" when using floating IPs on the native VLAN
  NeutronExternalNetworkBridge: """
```

If you use only one Floating IP network on the native VLAN of a bridge, you can optionally set the neutron external bridge. This results in the packets only having to traverse one bridge instead of two, which might result in slightly lower CPU usage when passing traffic over the Floating IP network.

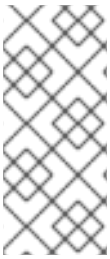
11.5. CONFIGURING THE NATIVE VLAN ON A TRUNKED INTERFACE

If a trunked interface or bond has a network on the native VLAN, the IP addresses are assigned directly to the bridge and is no VLAN interface.

For example, if the External network is on the native VLAN, a bonded configuration looks like this:

```
network_config:
  - type: ovs_bridge
    name: bridge_name
    dns_servers:
      get_param: DnsServers
    addresses:
      - ip_netmask:
          get_param: ExternalIpSubnet
    routes:
      - ip_netmask: 0.0.0.0/0
        next_hop:
          get_param: ExternalInterfaceDefaultRoute
    members:
      - type: ovs_bond
        name: bond1
        ovs_options:
          get_param: BondInterfaceOvsOptions
```

```
members:
  - type: interface
    name: nic3
    primary: true
  - type: interface
    name: nic4
```



NOTE

When moving the address (and possibly route) statements onto the bridge, remove the corresponding VLAN interface from the bridge. Make the changes to all applicable roles. The External network is only on the controllers, so only the controller template requires a change. The Storage network on the other hand is attached to all roles, so if the Storage network is on the default VLAN, all roles require modifications.

CHAPTER 12. NETWORK INTERFACE BONDING

This chapter defines some of the bonding options you can use in your custom network configuration.

12.1. NETWORK INTERFACE BONDING AND LINK AGGREGATION CONTROL PROTOCOL (LACP)

You can bundle multiple physical NICs together to form a single logical channel known as a bond. Bonds can be configured to provide redundancy for high availability systems or increased throughput.

Red Hat OpenStack Platform supports Linux bonds, Open vSwitch (OVS) kernel bonds, and OVS-DPDK bonds.

The bonds can be used with the optional Link Aggregation Control Protocol (LACP). LACP is a negotiation protocol that creates a dynamic bond for load balancing and fault tolerance.

On any network that interacts directly with virtual machine instances, Red Hat recommends the use of OVS kernel bonds (bond type ovs_bond) or OVS-DPDK bonds (bond type ovs_dpdk_bond) with LACP. However, do not combine OVS kernel bonds and OVS-DPDK bonds on the same node.

On control and storage networks, Red Hat recommends the use of Linux bonds with VLAN and LACP, because OVS bonds carry the potential for control plane disruption that can occur when OVS or the neutron agent is restarted for updates, hot fixes, and other events. The Linux bond/LACP/VLAN configuration provides NIC management without the OVS disruption potential. Here is an example configuration of a Linux bond with one VLAN.

```
params:
          $network_config:
            network_config:

            - type: linux_bond
              name: bond_api
              bonding_options: "mode=active-backup"
              use_dhcp: false
              dns_servers:
      `         get_param: DnsServers
              members:
              - type: interface
                name: nic3
                primary: true
              - type: interface
                name: nic4

            - type: vlan
              vlan_id:
                get_param: InternalApiNetworkVlanID
              device: bond_api
              addresses:
              - ip_netmask:
                  get_param: InternalApiIpSubnet
```

12.2. OPEN VSWITCH BONDING OPTIONS

The Overcloud provides networking through Open vSwitch (OVS). The following table describes support for OVS kernel and OVS-DPDK for bonded interfaces. The OVS/OVS-DPDK balance-tcp mode is available as a technology preview only.

NOTE

This support requires Open vSwitch 2.11 or later.

OVS Bond mode	Application	Notes	Compatible LACP options
active-backup	High availability (active-passive)		active, passive, or off
balance-slb	Increased throughput (active-active)	• Performance is affected by extra parsing per packet. • There is a potential for vhost-user lock contention.	active, passive, or off
balance-tcp (tech preview only)	Not recommended (active-active)	• Recirculation needed for L4 hashing has a performance impact. • As with balance-slb, performance is affected by extra parsing per packet and there is a potential for vhost-user lock contention. • LACP must be enabled.	active or passive

You can configure a bonded interface in the network environment file using the BondInterfaceOvsOptions parameter as shown in this example:

```
parameter_defaults:
  BondInterfaceOvsOptions: "bond_mode=balance-slb"
```

12.3. LINUX BONDING OPTIONS

You can use LACP with Linux bonding in your network interface templates. For example:

```
- type: linux_bond
  name: bond1
  members:
  - type: interface
    name: nic2
  - type: interface
    name: nic3
  bonding_options: "mode=802.3ad lacp_rate=[fast|slow] updelay=1000 miimon=100"
```

- **mode** - enables LACP.
- **lacp_rate** - defines whether LACP packets are sent every 1 second, or every 30 seconds.
- **updelay** - defines the minimum amount of time that an interface must be active before it is used for traffic (this helps mitigate port flapping outages).
- **miimon** - the interval in milliseconds that is used for monitoring the port state using the driver's MIIMON functionality.

12.4. GENERAL BONDING OPTIONS

The following table provides some explanation of these options and some alternatives depending on your hardware.

Table 12.1. Bonding Options

bond_mode=balance-slb	Balances flows based on source MAC address and output VLAN, with periodic rebalancing as traffic patterns change. Bonding with **balance-slb** allows a limited form of load balancing without the remote switch's knowledge or cooperation. SLB assigns each source MAC and VLAN pair to a link and transmits all packets from that MAC and VLAN through that link. This mode uses a simple hashing algorithm based on source MAC address and VLAN number, with periodic rebalancing as traffic patterns change. This mode is similar to mode 2 bonds used by the Linux bonding driver. This mode can be used to provide load balancing even when the switch is not configured to use LACP.
bond_mode=active-backup	This mode offers active/standby failover where the standby NIC resumes network operations when the active connection fails. Only one MAC address is presented to the physical switch. This mode does not require any special switch support or configuration, and works when the links are connected to separate switches. This mode does not provide load balancing.
lacp=[active|passive|off]	Controls the Link Aggregation Control Protocol (LACP) behavior. Only certain switches support LACP. If your switch does not support LACP, use **bond_mode=balance-slb** or **bond_mode=active-backup**.

other-config:lacp-fallback-ab=true	Sets the LACP behavior to switch to bond_mode=active-backup as a fallback.
other_config:lacp-time=[fast|slow]	Set the LACP heartbeat to 1 second (fast) or 30 seconds (slow). The default is slow.
other_config:bond-detect-mode=[miimon|carrier]	Set the link detection to use miimon heartbeats (miimon) or monitor carrier (carrier). The default is carrier.
other_config:bond-miimon-interval=100	If using miimon, set the heartbeat interval in milliseconds.
other_config:bond_updelay=1000	Number of milliseconds a link must be up to be activated to prevent flapping.
other_config:bond-rebalance-interval=10000	Milliseconds between rebalancing flows between bond members. Set to zero to disable.

CHAPTER 13. CONTROLLING NODE PLACEMENT

The default behavior for the director is to randomly select nodes for each role, usually based on their profile tag. However, the director provides the ability to define specific node placement. This is a useful method to:

- Assign specific node IDs e.g. **controller-0**, **controller-1**, etc
- Assign custom hostnames
- Assign specific IP addresses
- Assign specific Virtual IP addresses

NOTE

Manually setting predictable IP addresses, virtual IP addresses, and ports for a network alleviates the need for allocation pools. However, it is recommended to retain allocation pools for each network to ease with scaling new nodes. Make sure that any statically defined IP addresses fall outside the allocation pools. For more information on setting allocation pools, see Section 10.7, “Custom network environment file” .

13.1. ASSIGNING SPECIFIC NODE IDS

This procedure assigns node ID to specific nodes. Examples of node IDs include **controller-0**, **controller-1**, **compute-0**, **compute-1**, and so forth.

The first step is to assign the ID as a per-node capability that the Compute scheduler matches on deployment. For example:

```
openstack baremetal node set --property capabilities='node:controller-0,boot_option:local' <id>
```

This assigns the capability **node:controller-0** to the node. Repeat this pattern using a unique continuous index, starting from 0, for all nodes. Make sure all nodes for a given role (Controller, Compute, or each of the storage roles) are tagged in the same way or else the Compute scheduler will not match the capabilities correctly.

The next step is to create a Heat environment file (for example, **scheduler_hints_env.yaml**) that uses scheduler hints to match the capabilities for each node. For example:

```
parameter_defaults:
  ControllerSchedulerHints:
    'capabilities:node': 'controller-%index%'
```

To use these scheduler hints, include the ` scheduler_hints_env.yaml` environment file with the **overcloud deploy command** during Overcloud creation.

The same approach is possible for each role via these parameters:

- **ControllerSchedulerHints** for Controller nodes.
- **ComputeSchedulerHints** for Compute nodes.
- **BlockStorageSchedulerHints** for Block Storage nodes.

- **ObjectStorageSchedulerHints** for Object Storage nodes.
- **CephStorageSchedulerHints** for Ceph Storage nodes.
- **[ROLE]SchedulerHints** for custom roles. Replace **[ROLE]** with the role name.

NOTE

Node placement takes priority over profile matching. To avoid scheduling failures, use the default **baremetal** flavor for deployment and not the flavors designed for profile matching (**compute**, **control**, etc). For example:

```
$ openstack overcloud deploy ... --control-flavor baremetal --compute-flavor baremetal
...
```

13.2. ASSIGNING CUSTOM HOSTNAMES

In combination with the node ID configuration in Section 13.1, “Assigning Specific Node IDs” , the director can also assign a specific custom hostname to each node. This is useful when you need to define where a system is located (e.g. **rack2-row12**), match an inventory identifier, or other situations where a custom hostname is desired.

To customize node hostnames, use the **HostnameMap** parameter in an environment file, such as the ` scheduler_hints_env.yaml` file from Section 13.1, “Assigning Specific Node IDs” . For example:

```
parameter_defaults:
  ControllerSchedulerHints:
    'capabilities:node': 'controller-%index%'
  ComputeSchedulerHints:
    'capabilities:node': 'compute-%index%'
  HostnameMap:
    overcloud-controller-0: overcloud-controller-prod-123-0
    overcloud-controller-1: overcloud-controller-prod-456-0
    overcloud-controller-2: overcloud-controller-prod-789-0
    overcloud-compute-0: overcloud-compute-prod-abc-0
```

Define the **HostnameMap** in the **parameter_defaults** section, and set each mapping as the original hostname that Heat defines using **HostnameFormat** parameters (e.g. **overcloud-controller-0**) and the second value is the desired custom hostname for that node (e.g. **overcloud-controller-prod-123-0**).

Using this method in combination with the node ID placement ensures each node has a custom hostname.

13.3. ASSIGNING PREDICTABLE IPS

For further control over the resulting environment, the director can assign Overcloud nodes with specific IPs on each network as well. Use the **environments/ips-from-pool-all.yaml** environment file in the core Heat template collection. Copy this file to the **stack** user’s **templates** directory.

```
$ cp /usr/share/openstack-tripleo-heat-templates/environments/ips-from-pool-all.yaml ~/templates/.
```

There are two major sections in the **ips-from-pool-all.yaml** file.

The first is a set of **resource_registry** references that override the defaults. These tell the director to use a specific IP for a given port on a node type. Modify each resource to use the absolute path of its respective template. For example:

```
  OS::TripleO::Controller::Ports::ExternalPort: /usr/share/openstack-tripleo-heat-
templates/network/ports/external_from_pool.yaml
  OS::TripleO::Controller::Ports::InternalApiPort: /usr/share/openstack-tripleo-heat-
templates/network/ports/internal_api_from_pool.yaml
  OS::TripleO::Controller::Ports::StoragePort: /usr/share/openstack-tripleo-heat-
templates/network/ports/storage_from_pool.yaml
  OS::TripleO::Controller::Ports::StorageMgmtPort: /usr/share/openstack-tripleo-heat-
templates/network/ports/storage_mgmt_from_pool.yaml
  OS::TripleO::Controller::Ports::TenantPort: /usr/share/openstack-tripleo-heat-
templates/network/ports/tenant_from_pool.yaml
```

The default configuration sets all networks on all node types to use pre-assigned IPs. To allow a particular network or node type to use default IP assignment instead, simply remove the **resource_registry** entries related to that node type or network from the environment file.

The second section is parameter_defaults, where the actual IP addresses are assigned. Each node type has an associated parameter:

- **ControllerIPs** for Controller nodes.
- **ComputeIPs** for Compute nodes.
- **CephStorageIPs** for Ceph Storage nodes.
- **BlockStorageIPs** for Block Storage nodes.
- **SwiftStorageIPs** for Object Storage nodes.
- **[ROLE]IPs** for custom roles. Replace **[ROLE]** with the role name.

Each parameter is a map of network names to a list of addresses. Each network type must have at least as many addresses as there will be nodes on that network. The director assigns addresses in order. The first node of each type receives the first address on each respective list, the second node receives the second address on each respective lists, and so forth.

For example, if an Overcloud will contain three Ceph Storage nodes, the **CephStorageIPs** parameter might look like:

```
CephStorageIPs:
  storage:
  - 172.16.1.100
  - 172.16.1.101
  - 172.16.1.102
  storage_mgmt:
  - 172.16.3.100
  - 172.16.3.101
  - 172.16.3.102
```

The first Ceph Storage node receives two addresses: 172.16.1.100 and 172.16.3.100. The second receives 172.16.1.101 and 172.16.3.101, and the third receives 172.16.1.102 and 172.16.3.102. The same pattern applies to the other node types.

To configure predictable IP addresses on the control plane, copy the **/usr/share/openstack-tripleo-heat-templates/environments/ips-from-pool-ctlplane.yaml** file to the **templates** directory of the **stack** user:

```
$ cp /usr/share/openstack-tripleo-heat-templates/environments/ips-from-pool-ctlplane.yaml
~/templates/.
```

Configure the new **ips-from-pool-ctlplane.yaml** file with the following parameter example. You can combine the control plane IP address declarations with the IP address declarations for other networks and use only one file to declare the IP addresses for all networks on all roles. You can also use predictable IP addresses for spine/leaf. Each node must have IP addresses from the correct subnet.

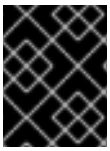
```
parameter_defaults:
  ControllerIPs:
    ctlplane:
    - 192.168.24.10
    - 192.168.24.11
    - 192.168.24.12
    internal_api:
    - 172.16.1.20
    - 172.16.1.21
    - 172.16.1.22
    external:
    - 10.0.0.40
    - 10.0.0.57
    - 10.0.0.104
  ComputeLeaf1IPs:
    ctlplane:
    - 192.168.25.100
    - 192.168.25.101
    internal_api:
    - 172.16.2.100
    - 172.16.2.101
  ComputeLeaf2IPs:
    ctlplane:
    - 192.168.26.100
    - 192.168.26.101
    internal_api:
    - 172.16.3.100
    - 172.16.3.101
```

Make sure the chosen IP addresses fall outside the allocation pools for each network defined in your network environment file (see Section 10.7, “Custom network environment file”). For example, make sure the **internal_api** assignments fall outside of the **InternalApiAllocationPools** range. This avoids conflicts with any IPs chosen automatically. Likewise, make sure the IP assignments do not conflict with the VIP configuration, either for standard predictable VIP placement (see Section 13.4, “Assigning Predictable Virtual IPs”) or external load balancing (see Section 24.1, “Configuring External Load Balancing”).

IMPORTANT

If an overcloud node is deleted, do not remove its entries in the IP lists. The IP list is based on the underlying Heat indices, which do not change even if you delete nodes. To indicate a given entry in the list is no longer used, replace the IP value with a value such as **DELETED** or **UNUSED**. Entries should never be removed from the IP lists, only changed or added.

To apply this configuration during a deployment, include the **ips-from-pool-all.yaml** environment file with the **openstack overcloud deploy** command.



IMPORTANT

If using network isolation, include the **ips-from-pool-all.yaml** file after the **network-isolation.yaml** file.

For example:

```
$ openstack overcloud deploy --templates \
  -e /usr/share/openstack-tripleo-heat-templates/environments/network-isolation.yaml \
  -e ~/templates/ips-from-pool-all.yaml \
  [OTHER OPTIONS]
```

13.4. ASSIGNING PREDICTABLE VIRTUAL IPS

In addition to defining predictable IP addresses for each node, the director also provides a similar ability to define predictable Virtual IPs (VIPs) for clustered services. To accomplish this, edit the network environment file from Section 10.7, "Custom network environment file" and add the VIP parameters in the **parameter_defaults** section:

```
parameter_defaults:
  ...
  # Predictable VIPs
  ControlFixedIPs: [{'ip_address':'192.168.201.101'}]
  InternalApiVirtualFixedIPs: [{'ip_address':'172.16.0.9'}]
  PublicVirtualFixedIPs: [{'ip_address':'10.1.1.9'}]
  StorageVirtualFixedIPs: [{'ip_address':'172.18.0.9'}]
  StorageMgmtVirtualFixedIPs: [{'ip_address':'172.19.0.9'}]
  RedisVirtualFixedIPs: [{'ip_address':'172.16.0.8'}]
```

Select these IPs from outside of their respective allocation pool ranges. For example, select an IP address for **InternalApiVirtualFixedIPs** that is not within the **InternalApiAllocationPools** range.

This step is only for overclouds using the default internal load balancing configuration. If assigning VIPs with an external load balancer, use the procedure in the dedicated External Load Balancing for the Overcloud guide.

CHAPTER 14. ENABLING SSL/TLS ON OVERCLOUD PUBLIC ENDPOINTS

By default, the overcloud uses unencrypted endpoints for its services. This means that the overcloud configuration requires an additional environment file to enable SSL/TLS for its Public API endpoints. The following chapter shows how to configure your SSL/TLS certificate and include it as a part of your overcloud creation.

NOTE

This process only enables SSL/TLS for Public API endpoints. The Internal and Admin APIs remain unencrypted.

This process requires network isolation to define the endpoints for the Public API.

14.1. INITIALIZING THE SIGNING HOST

The signing host is the host that generates and signs new certificates with a certificate authority. If you have never created SSL certificates on the chosen signing host, you might need to initialize the host so that it can sign new certificates.

The **/etc/pki/CA/index.txt** file contains records of all signed certificates. Check if this file exists. If it does not exist, create an empty file:

```
$ sudo touch /etc/pki/CA/index.txt
```

The **/etc/pki/CA/serial** file identifies the next serial number to use for the next certificate to sign. Check if this file exists. If the file does not exist, create a new file with a new starting value:

```
$ echo '1000' | sudo tee /etc/pki/CA/serial
```

14.2. CREATING A CERTIFICATE AUTHORITY

Normally you sign your SSL/TLS certificates with an external certificate authority. In some situations, you might want to use your own certificate authority. For example, you might want to have an internal-only certificate authority.

Generate a key and certificate pair to act as the certificate authority:

```
$ openssl genrsa -out ca.key.pem 4096
$ openssl req  -key ca.key.pem -new -x509 -days 7300 -extensions v3_ca -out ca.crt.pem
```

The **openssl req** command asks for certain details about your authority. Enter these details at the prompt.

These commands create a certificate authority file called **ca.crt.pem**.

14.3. ADDING THE CERTIFICATE AUTHORITY TO CLIENTS

For any external clients aiming to communicate using SSL/TLS, copy the certificate authority file to each client that requires access to your Red Hat OpenStack Platform environment.

```
$ sudo cp ca.crt.pem /etc/pki/ca-trust/source/anchors/
```

After you copy the certificate authority file to each client, run the following command on each client to add the certificate to the certificate authority trust bundle:

```
$ sudo update-ca-trust extract
```

For example, the undercloud requires a copy of the certificate authority file so that it can communicate with the overcloud endpoints during creation.

14.4. CREATING AN SSL/TLS KEY

Run the following commands to generate the SSL/TLS key (**server.key.pem**) that you use at different points to generate your undercloud or overcloud certificates:

```
$ openssl genrsa -out server.key.pem 2048
```

14.5. CREATING AN SSL/TLS CERTIFICATE SIGNING REQUEST

This next procedure creates a certificate signing request for the overcloud. Copy the default OpenSSL configuration file for customization.

```
$ cp /etc/pki/tls/openssl.cnf .
```

Edit the custom **openssl.cnf** file and set SSL parameters to use for the overcloud. An example of the types of parameters to modify include:

```
[req]
distinguished_name = req_distinguished_name
req_extensions = v3_req

[req_distinguished_name]
countryName = Country Name (2 letter code)
countryName_default = AU
stateOrProvinceName = State or Province Name (full name)
stateOrProvinceName_default = Queensland
localityName = Locality Name (eg, city)
localityName_default = Brisbane
organizationalUnitName = Organizational Unit Name (eg, section)
organizationalUnitName_default = Red Hat
commonName = Common Name
commonName_default = 10.0.0.1
commonName_max = 64

[ v3_req ]
# Extensions to add to a certificate request
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
subjectAltName = @alt_names

[alt_names]
```

```
IP.1 = 10.0.0.1
DNS.1 = 10.0.0.1
DNS.2 = myovercloud.example.com
```

Set the **commonName_default** to one of the following:

- If using an IP to access over SSL/TLS, use the Virtual IP for the Public API. Set this VIP using the **PublicVirtualFixedIPs** parameter in an environment file. For more information, see Section 13.4, "Assigning Predictable Virtual IPs" . If you are not using predictable VIPs, the director assigns the first IP address from the range defined in the **ExternalAllocationPools** parameter.
- If using a fully qualified domain name to access over SSL/TLS, use the domain name instead.

Include the same Public API IP address as an IP entry and a DNS entry in the **alt_names** section. If also using DNS, include the hostname for the server as DNS entries in the same section. For more information about **openssl.cnf**, run **man openssl.cnf**.

Run the following command to generate certificate signing request (**server.csr.pem**):

```
$ openssl req -config openssl.cnf -key server.key.pem -new -out server.csr.pem
```

Make sure to include the SSL/TLS key you created in Section 14.4, "Creating an SSL/TLS Key" for the **-key** option.

Use the **server.csr.pem** file to create the SSL/TLS certificate in the next section.

14.6. CREATING THE SSL/TLS CERTIFICATE

Run the following command to create a certificate for your undercloud or overcloud:

```
$ sudo openssl ca -config openssl.cnf -extensions v3_req -days 3650 -in server.csr.pem -out
server.crt.pem -cert ca.crt.pem -keyfile ca.key.pem
```

This command uses the following options:

- The configuration file specifying the v3 extensions. Include the configuration file with the **-config** option.
- The certificate signing request from Section 14.5, "Creating an SSL/TLS Certificate Signing Request" to generate and sign the certificate with a certificate authority. Include the certificate signing request with the **-in** option.
- The certificate authority you created in Section 14.2, "Creating a Certificate Authority" , which signs the certificate. Include the certificate authority with the **-cert** option.
- The certificate authority private key you created in Section 14.2, "Creating a Certificate Authority". Include the private key with the **-keyfile** option.

This command creates a new certificate named **server.crt.pem**. Use this certificate in conjunction with the SSL/TLS key from Section 14.4, "Creating an SSL/TLS Key" to enable SSL/TLS.

14.7. ENABLING SSL/TLS

Copy the **enable-tls.yaml** environment file from the Heat template collection:

```
$ cp -r /usr/share/openstack-tripleo-heat-templates/environments/ssl/enable-tls.yaml ~/templates/.
```

Edit this file and make the following changes for these parameters:

SSLCertificate

Copy the contents of the certificate file (**server.crt.pem**) into the **SSLCertificate** parameter. For example:

```
parameter_defaults:
  SSLCertificate: |
    -----BEGIN CERTIFICATE-----
    MIIDgzCCAmugAwIBAgIJAKk46qw6ncJaMA0GCSqGS
    ...
    sFW3S2roS4X0Af/kSSD8mlBBTFTCMBAj6rtLBKLaQ
    -----END CERTIFICATE-----
```

IMPORTANT

The certificate contents require the same indentation level for all new lines.

SSLKey

Copy the contents of the private key (**server.key.pem**) into the **SSLKey** parameter. For example:

```
parameter_defaults:
  ...
  SSLKey: |
    -----BEGIN RSA PRIVATE KEY-----
    MIIEowIBAAKCAQEAqVw8lnQ9RbeI1EdLN5PJP0lVO
    ...
    ctlKn3rAAdyumi4JDjESAXHIKFjJNOLrBmpQyES4X
    -----END RSA PRIVATE KEY-----
```

IMPORTANT

The private key contents require the same indentation level for all new lines.

14.8. INJECTING A ROOT CERTIFICATE

If the certificate signer is not in the default trust store on the overcloud image, you must inject the certificate authority into the overcloud image. Copy the **inject-trust-anchor-hiera.yaml** environment file from the heat template collection:

```
$ cp -r /usr/share/openstack-tripleo-heat-templates/environments/ssl/inject-trust-anchor-hiera.yaml
~/templates/.
```

Edit this file and make the following changes for these parameters:

CAMap

Lists each certificate authority content (CA) to inject into the overcloud. The overcloud requires both a CA files used to sign the certificates for the undercloud and the overcloud. Copy the contents of the root certificate authority file (**ca.crt.pem**) into an entry. For example, your **CAMap** parameter might look like the following:

```
parameter_defaults:
  CAMap:
    ...
    undercloud-ca:
      content: |
        -----BEGIN CERTIFICATE-----
        MIIDITCCAn2gAwIBAgIJAOnPtx2hHEhrMA0GCS
        BAYTAlVTMQswCQYDVQQIDAJOQzEQMA4GA1UEBw
        UmVkIEhhdDELMAkGA1UECwwCUUUxFDASBgNVBA
        -----END CERTIFICATE-----
    overcloud-ca:
      content: |
        -----BEGIN CERTIFICATE-----
        MIIDBzCCAe+gAwIBAgIJAIc75A7FD++DMA0GCS
        BAMMD3d3dy5leGFtcGxlLmNvbTAeFw0xOTAxMz
        Um54yGCARyp3LpkxvyfMXX1DokpS1uKi7s6CkF
        -----END CERTIFICATE-----
```

IMPORTANT

The certificate authority contents require the same indentation level for all new lines.

You can also inject additional CAs with the **CAMap** parameter.

14.9. CONFIGURING DNS ENDPOINTS

If using a DNS hostname to access the overcloud through SSL/TLS, create a new environment file (**~/templates/cloudname.yaml**) to define the hostname of the overcloud's endpoints. Use the following parameters:

CloudName

The DNS hostname of the overcloud endpoints.

DnsServers

A list of DNS servers to use. The configured DNS servers must contain an entry for the configured **CloudName** that matches the IP address of the Public API.

An example of the contents for this file:

```
parameter_defaults:
  CloudName: overcloud.example.com
  DnsServers: ["10.0.0.254"]
```

14.10. ADDING ENVIRONMENT FILES DURING OVERCLOUD CREATION

The deployment command (**openstack overcloud deploy**) uses the **-e** option to add environment files. Add the environment files from this section in the following order:

- The environment file to enable SSL/TLS (**enable-tls.yaml**)
- The environment file to set the DNS hostname (**cloudname.yaml**)
- The environment file to inject the root certificate authority (**inject-trust-anchor-hiera.yaml**)
- The environment file to set the public endpoint mapping:
  - If using a DNS name for accessing the public endpoints, use **/usr/share/openstack-tripleo-heat-templates/environments/ssl/tls-endpoints-public-dns.yaml**
  - If using a IP address for accessing the public endpoints, use **/usr/share/openstack-tripleo-heat-templates/environments/ssl/tls-endpoints-public-ip.yaml**

For example:

```
$ openstack overcloud deploy --templates [...] -e /home/stack/templates/enable-tls.yaml -e ~/templates/cloudname.yaml -e ~/templates/inject-trust-anchor-hiera.yaml -e /usr/share/openstack-tripleo-heat-templates/environments/ssl/tls-endpoints-public-dns.yaml
```

14.11. UPDATING SSL/TLS CERTIFICATES

If you need to update certificates in the future:

- Edit the **enable-tls.yaml** file and update the **SSLCertificate**, **SSLKey**, and **SSLIntermediateCertificate** parameters.
- If your certificate authority has changed, edit the **inject-trust-anchor.yaml** file and update the **SSLRootCertificate** parameter.

Once the new certificate content is in place, rerun your deployment command. For example:

```
$ openstack overcloud deploy --templates [...] -e /home/stack/templates/enable-tls.yaml -e ~/templates/cloudname.yaml -e ~/templates/inject-trust-anchor.yaml -e /usr/share/openstack-tripleo-heat-templates/environments/ssl/tls-endpoints-public-dns.yaml
```

CHAPTER 15. ENABLING SSL/TLS ON INTERNAL AND PUBLIC ENDPOINTS WITH IDENTITY MANAGEMENT

You can enable SSL/TLS on certain overcloud endpoints. Due to the number of certificates required, the director integrates with a Red Hat Identity Management (IdM) server to act as a certificate authority and manage the overcloud certificates. This process involves using **novajoin** to enroll overcloud nodes to the IdM server.

15.1. ADD THE UNDERCLOUD TO THE CA

Before deploying the overcloud, you must add the undercloud to the Certificate Authority (CA):

1. On the undercloud node, install the **python-novajoin** package:

```
$ sudo yum install python-novajoin
```

2. On the undercloud node, run the **novajoin-ipa-setup** script, adjusting the values to suit your deployment:

```
$ sudo /usr/libexec/novajoin-ipa-setup \
  --principal admin \
  --password <IdM admin password> \
  --server <IdM server hostname> \
  --realm <overcloud cloud domain (in upper case)> \
  --domain <overcloud cloud domain> \
  --hostname <undercloud hostname> \
  --precreate
```

   In the following section, you will use the resulting One-Time Password (OTP) to enroll the undercloud.

15.2. ADD THE UNDERCLOUD TO IDM

This procedure registers the undercloud with IdM and configures novajoin. Configure the following settings in **undercloud.conf** (within the **[DEFAULT]** section):

1. The novajoin service is disabled by default. To enable it:

```
[DEFAULT]
enable_novajoin = true
```

2. You need set a One-Time Password (OTP) to register the undercloud node with IdM:

```
ipa_otp = <otp>
```

3. Ensure the overcloud's domain name served by neutron's DHCP server matches the IdM domain (your kerberos realm in lowercase):

```
overcloud_domain_name = <domain>
```

4. Set the appropriate hostname for the undercloud:

```
undercloud_hostname = <undercloud FQDN>
```

5. Set IdM as the nameserver for the undercloud:

```
undercloud_nameservers = <IdM IP>
```

6. For larger environments, you will need to review the novajoin connection timeout values. In **undercloud.conf**, add a reference to a new file called **undercloud-timeout.yaml**:

```
hieradata_override = /home/stack/undercloud-timeout.yaml
```

Add the following options to **undercloud-timeout.yaml**. You can specify the timeout value in seconds, for example, **5**:

```
nova::api::vendordata_dynamic_connect_timeout: <timeout value>
nova::api::vendordata_dynamic_read_timeout: <timeout value>
```

7. Save the **undercloud.conf** file.

8. Run the undercloud deployment command to apply the changes to your existing undercloud:

```
$ openstack undercloud install
```

15.3. CONFIGURE OVERCLOUD DNS

For automatic detection of your IdM environment, and easier enrollment, consider using IdM as your DNS server:

1. Connect to your undercloud:

```
$ source ~/stackrc
```

2. Configure the control plane subnet to use IdM as the DNS name server:

```
$ openstack subnet set ctlplane-subnet --dns-nameserver  <idm_server_address>
```

3. Set the **DnsServers** parameter in an environment file to use your IdM server:

```
parameter_defaults:
  DnsServers: ["<idm_server_address>"]
```

This parameter is usually defined in a custom **network-environment.yaml** file.

15.4. CONFIGURE OVERCLOUD TO USE NOVAJOIN

1. To enable IdM integration, create a copy of the **/usr/share/openstack-tripleo-heat-templates/environments/predictable-placement/custom-domain.yaml** environment file:

```
$ cp /usr/share/openstack-tripleo-heat-templates/environments/predictable-
placement/custom-domain.yaml \
  /home/stack/templates/custom-domain.yaml
```

2. Edit the **/home/stack/templates/custom-domain.yaml** environment file and set the **CloudDomain** and **CloudName*** values to suit your deployment. For example:

```
parameter_defaults:
  CloudDomain: lab.local
  CloudName: overcloud.lab.local
  CloudNameInternal: overcloud.internalapi.lab.local
  CloudNameStorage: overcloud.storage.lab.local
  CloudNameStorageManagement: overcloud.storagemgmt.lab.local
  CloudNameCtlplane: overcloud.ctlplane.lab.local
```

3. Include the following environment files in the overcloud deployment process:

   - **/usr/share/openstack-tripleo-heat-templates/environments/ssl/enable-internal-tls.yaml**
   - **/usr/share/openstack-tripleo-heat-templates/environments/ssl/tls-everywhere-endpoints-dns.yaml**
   - **/home/stack/templates/custom-domain.yaml**
     For example:

```
openstack overcloud deploy \
  --templates \
    -e /usr/share/openstack-tripleo-heat-templates/environments/ssl/enable-internal-tls.yaml \
    -e /usr/share/openstack-tripleo-heat-templates/environments/ssl/tls-everywhere-endpoints-dns.yaml \
    -e /home/stack/templates/custom-domain.yaml \
```

     As a result, the deployed overcloud nodes will be automatically enrolled with IdM.

4. This only sets TLS for the internal endpoints. For the external endpoints you can use the normal means of adding TLS with the **/usr/share/openstack-tripleo-heat-templates/environments/ssl/enable-tls.yaml** environment file (which must be modified to add your custom certificate and key). Consequently, your **openstack deploy** command would be similar to this:

```
openstack overcloud deploy \
  --templates \
  -e /usr/share/openstack-tripleo-heat-templates/environments/ssl/enable-internal-tls.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/ssl/tls-everywhere-endpoints-dns.yaml \
  -e /home/stack/templates/custom-domain.yaml \
  -e /home/stack/templates/enable-tls.yaml
```

5. Alternatively, you can also use IdM to issue your public certificates. In that case, you need to use the **/usr/share/openstack-tripleo-heat-templates/environments/services/haproxy-public-tls-certmonger.yaml** environment file. For example:

```
openstack overcloud deploy \
  --templates \
    -e /usr/share/openstack-tripleo-heat-templates/environments/ssl/enable-internal-tls.yaml \
    -e /usr/share/openstack-tripleo-heat-templates/environments/ssl/tls-everywhere-endpoints-dns.yaml \
```

```
  -e /home/stack/templates/custom-domain.yaml \
  -e /usr/share/openstack-tripleo-heat-templates/environments/services/haproxy-public-tls-
certmonger.yaml
```

CHAPTER 16. DEBUG MODES

You can enable and disable the **DEBUG** level logging mode for certain services in the overcloud. To configure debug mode for a service, set the respective debug parameter. For example, OpenStack Identity (keystone) uses the **KeystoneDebug** parameter. Set this parameter in the **parameter_defaults** section of an environment file:

```
parameter_defaults:
  KeystoneDebug: True
```

For a full list of debug parameters, see "Debug Parameters" in the *Overcloud Parameters* guide.

CHAPTER 17. POLICIES

You can configure access policies for certain services in the overcloud. To configure policies for a service, set the respective policy parameter with a hash value containing the service's policies. For example, OpenStack Identity (keystone) uses the **KeystonePolicies** parameter. Set this parameter in the **parameter_defaults** section of an environment file:

```
parameter_defaults:
  KeystonePolicies: { keystone-context_is_admin: { key: context_is_admin, value: 'role:admin' } }
```

For a full list of policy parameters, see "Policy Parameters" in the *Overcloud Parameters* guide.

CHAPTER 18. STORAGE CONFIGURATION

This chapter outlines several methods of configuring storage options for your Overcloud.

IMPORTANT

The Overcloud uses local and LVM storage for the default storage options. However, these options are not supported for enterprise-level Overclouds. It is recommended to use one of the storage options in this chapter.

18.1. CONFIGURING NFS STORAGE

This section describes configuring the Overcloud to use an NFS share. The installation and configuration process is based on the modification of an existing environment file in the core Heat template collection.

The core heat template collection contains a set of environment files in **/usr/share/openstack-tripleo-heat-templates/environments/**. These environment templates help with custom configuration of some of the supported features in a director-created Overcloud. This includes an environment file to help configure storage. This file is located at **/usr/share/openstack-tripleo-heat-templates/environments/storage-environment.yaml**. Copy this file to the **stack** user's template directory.

```
$ cp /usr/share/openstack-tripleo-heat-templates/environments/storage-environment.yaml
~/templates/.
```

The environment file contains some parameters to help configure different storage options for OpenStack's block and image storage components, cinder and glance. In this example, you will configure the Overcloud to use an NFS share. Modify the following parameters:

CinderEnableIscsiBackend

Enables the iSCSI backend. Set to **false**.

CinderEnableRbdBackend

Enables the Ceph Storage backend. Set to **false**.

CinderEnableNfsBackend

Enables the NFS backend. Set to **true**.

NovaEnableRbdBackend

Enables Ceph Storage for Nova ephemeral storage. Set to **false**.

GlanceBackend

Define the back end to use for Glance. Set to **file** to use file-based storage for images. The Overcloud will save these files in a mounted NFS share for Glance.

CinderNfsMountOptions

The NFS mount options for the volume storage.

CinderNfsServers

The NFS share to mount for volume storage. For example, 192.168.122.1:/export/cinder.

GlanceNfsEnabled

Enables Pacemaker to manage the share for image storage. If disabled, the Overcloud stores images in the Controller node's file system. Set to **true**.

GlanceNfsShare

The NFS share to mount for image storage. For example, 192.168.122.1:/export/glance.

GlanceNfsOptions

The NFS mount options for the image storage.

The environment file's options should look similar to the following:

```
parameter_defaults:
  CinderEnableIscsiBackend: false
  CinderEnableRbdBackend: false
  CinderEnableNfsBackend: true
  NovaEnableRbdBackend: false
  GlanceBackend: 'file'

  CinderNfsMountOptions: 'rw,sync'
  CinderNfsServers: '192.0.2.230:/cinder'

  GlanceNfsEnabled: true
  GlanceNfsShare: '192.0.2.230:/glance'
  GlanceNfsOptions: 'rw,sync,context=system_u:object_r:glance_var_lib_t:s0'
```

IMPORTANT

Include the **context=system_u:object_r:glance_var_lib_t:s0** in the **GlanceNfsOptions** parameter to allow glance access to the **/var/lib** directory. Without this SELinux content, glance will fail to write to the mount point.

These parameters are integrated as part of the heat template collection. Setting them as such creates two NFS mount points for cinder and glance to use.

Save this file for inclusion in the Overcloud creation.

18.2. CONFIGURING CEPH STORAGE

The director provides two main methods for integrating Red Hat Ceph Storage into an Overcloud.

Creating an Overcloud with its own Ceph Storage Cluster

The director has the ability to create a Ceph Storage Cluster during the creation on the Overcloud. The director creates a set of Ceph Storage nodes that use the Ceph OSD to store the data. In addition, the director install the Ceph Monitor service on the Overcloud's Controller nodes. This means if an organization creates an Overcloud with three highly available controller nodes, the Ceph Monitor also becomes a highly available service. For more information, see the Deploying an Overcloud with Containerized Red Hat Ceph guide.

Integrating a Existing Ceph Storage into an Overcloud

If you already have an existing Ceph Storage Cluster, you can integrate this during an Overcloud deployment. This means you manage and scale the cluster outside of the Overcloud configuration. For more information, see the Integrating an Overcloud with an Existing Red Hat Ceph Cluster guide.

18.3. USING AN EXTERNAL OBJECT STORAGE CLUSTER

You can reuse an external Object Storage (swift) cluster by disabling the default Object Storage service deployment on the controller nodes. Doing so disables both the proxy and storage services for Object Storage and configures haproxy and keystone to use the given external Swift endpoint.

NOTE

User accounts on the external Object Storage (swift) cluster have to be managed by hand.

You need the endpoint IP address of the external Object Storage cluster as well as the **authtoken** password from the external Object Storage **proxy-server.conf** file. You can find this information by using the **openstack endpoint list** command.

To deploy director with an external Swift cluster:

1. Create a new file named **swift-external-params.yaml** with the following content:
   - Replace **EXTERNAL.IP:PORT** with the IP address and port of the external proxy and
   - Replace **AUTHTOKEN** with the **authtoken** password for the external proxy on the **SwiftPassword** line.

```
parameter_defaults:
  ExternalPublicUrl: 'https://EXTERNAL.IP:PORT/v1/AUTH_%(tenant_id)s'
  ExternalInternalUrl: 'http://192.168.24.9:8080/v1/AUTH_%(tenant_id)s'
  ExternalAdminUrl: 'http://192.168.24.9:8080'
  ExternalSwiftUserTenant: 'service'
  SwiftPassword: AUTHTOKEN
```

2. Save this file as **swift-external-params.yaml**.
3. Deploy the overcloud using these additional environment files.

```
openstack overcloud deploy --templates \
-e [your environment files]
-e /usr/share/openstack-tripleo-heat-templates/environments/swift-external.yaml
-e swift-external-params.yaml
```

18.4. CONFIGURING THE IMAGE IMPORT METHOD AND SHARED STAGING AREA

The default settings for the OpenStack Image service (glance) are determined by the Heat templates used when OpenStack is installed. The Image service Heat template is **deployment/glance/glance-api-container-puppet.yaml**.

The interoperable image import allows two methods for image import:

- web-download
- glance-direct

The **web-download** method lets you import an image from a URL; the **glance-direct** method lets you import an image from a local volume.

18.4.1. Creating and Deploying the glance-settings.yaml File

You use an environment file to configure the import parameters. These parameters override the default values established in the Heat template. The example environment content provides parameters for the interoperable image import.

```
parameter_defaults:
  # Configure NFS backend
  GlanceBackend: file
  GlanceNfsEnabled: true
  GlanceNfsShare: 192.168.122.1:/export/glance

  # Enable glance-direct import method
  GlanceEnabledImportMethods: glance-direct,web-download

  # Configure NFS staging area (required for glance-direct import method)
  GlanceStagingNfsShare: 192.168.122.1:/export/glance-staging
```

The **GlanceBackend**, **GlanceNfsEnabled**, and **GlanceNfsShare** parameters are defined in the Storage Configuration section in the *Advanced Overcloud Customization Guide*.

Two new parameters for interoperable image import define the import method and a shared NFS staging area.

GlanceEnabledImportMethods

Defines the available import methods, web-download (default) and glance-direct. This line is only necessary if you wish to enable additional methods besides web-download.

GlanceStagingNfsShare

Configures the NFS staging area used by the glance-direct import method. This space can be shared amongst nodes in a high-availability cluster setup. Requires GlanceNfsEnabled be set to true.

To configure the settings:

1. Create a new file called, for example, glance-settings.yaml. The contents of this file should be similar to the example above.
2. Add the file to your OpenStack environment using the **openstack overcloud deploy** command:

```
$ openstack overcloud deploy --templates -e glance-settings.yaml
```

For additional information about using environment files, see the Including Environment Files in Overcloud Creation section in the *Advanced Overcloud Customization Guide*.

18.4.2. Controlling Image Web-Import Sources

You can limit the sources of Web-import image downloads by adding URI blacklists and whitelists to the optional **glance-image-import.conf** file.

You can whitelist or blacklist image source URIs at three levels:

- scheme (allowed_schemes, disallowed_schemes)
- host (allowed_hosts, disallowed_hosts)
- port (allowed_ports, disallowed_ports)

If you specify both at any level, the whitelist is honored and the blacklist is ignored.

The Image service applies the following decision logic to validate image source URIs:

1. The scheme is checked.
    a. Missing scheme: reject
    b. If there's a whitelist, and the scheme is not in it: reject. Otherwise, skip C and continue on to 2.
    c. If there's a blacklist, and the scheme is in it: reject.
2. The host name is checked.
    a. Missing host name: reject
    b. If there's a whitelist, and the host name is not in it: reject. Otherwise, skip C and continue on to 3.
    c. If there's a blacklist, and the host name is in it: reject.
3. If there's a port in the URI, the port is checked.
    a. If there's a whitelist, and the port is not in it: reject. Otherwise, skip B and continue on to 4.
    b. If there's a black list, and the port is in it: reject.
4. The URI is accepted as valid.

Note that if you allow a scheme, either by whitelisting it or by not blacklisting it, any URI that uses the default port for that scheme by not including a port in the URI is allowed. If it does include a port in the URI, the URI is validated according to the above rules.

18.4.2.1. Example

For instance, the default port for FTP is 21. Because *ftp* is a whitelisted scheme, this URL is allowed: ftp://example.org/some/resource But because 21 is not in the port whitelist, this URL to the same resource is rejected: ftp://example.org:21/some/resource

```
allowed_schemes = [http,https,ftp]
disallowed_schemes = []
allowed_hosts = []
disallowed_hosts = []
allowed_ports = [80,443]
disallowed_ports = []
```

[Including Environment Files in Overcloud Creation] section in the *Advanced Overcloud Customization Guide*.

18.4.2.2. Default Image Import Blacklist and Whitelist Settings

The glance-image-import.conf file is an optional file. Here are the default settings for these options:

- allowed_schemes - [*http*, *https*]
- disallowed_schemes - empty list
- allowed_hosts - empty list

- disallowed_hosts - empty list
- allowed_ports - [80, 443]
- disallowed_ports - empty list

Thus if you use the defaults, end users will only be able to access URIs using the *http* or *https* scheme. The only ports users will be able to specify are 80 and 443. (Users do not have to specify a port, but if they do, it must be either 80 or 443.)

You can find the **glance-image-import.conf** file in the etc/ subdirectory of the Image service source code tree. Make sure that you are looking in the correct branch for the OpenStack release you are working with.

18.4.3. Injecting Metadata on Image Import to Control Where VMs Launch

End users can add images in the Image service and use these images to launch VMs. These user-provided (non-admin) images should be launched on a specific set of compute nodes. The assignment of an instance to a compute node is controlled by image metadata properties.

The Image Property Injection plugin injects metadata properties to images on import. Specify the properties by editing the [image_import_opts] and [inject_metadata_properties] sections of the **glance-image-import.conf** file.

To enable the Image Property Injection plugin, add this line to the [image_import_opts] section:

```
[image_import_opts]
image_import_plugins = [inject_image_metadata]
```

To limit the metadata injection to images provided by a certain set of users, set the ignore_user_roles parameter. For instance, the following configuration injects one value for property1 and two values for property2 into images downloaded by by any non-admin user.

```
[DEFAULT]
[image_conversion]
[image_import_opts]
image_import_plugins = [inject_image_metadata]
[import_filtering_opts]
[inject_metadata_properties]
ignore_user_roles = admin
inject = PROPERTY1:value,PROPERTY2:value;another value
```

The parameter **ignore_user_roles** is a comma-separated list of Keystone roles that the plugin will ignore. In other words, if the user making the image import call has any of these roles, the plugin will not inject any properties into the image.

The parameter **inject** is a comma-separated list of properties and values that will be injected into the image record for the imported image. Each property and value should be quoted and separated by a colon (':') as shown in the example above.

You can find the **glance-image-import.conf** file in the etc/ subdirectory of the Image service source code tree. Make sure that you are looking in the correct branch for the OpenStack release you are working with.

18.5. CONFIGURING CINDER BACK END FOR THE IMAGE SERVICE

The **GlanceBackend** parameter sets the back end that the Image service uses to store images. To configure **cinder** as the Image service back end, add the following to the environment file:

```
parameter_defaults:
  GlanceBackend: cinder
```

If the **cinder** back end is enabled, the following parameters and values are set by default:

```
cinder_store_auth_address = http://172.17.1.19:5000/v3
cinder_store_project_name = service
cinder_store_user_name = glance
cinder_store_password = ****secret****
```

To use a custom user name, or any custom value for the **cinder_store_** parameters, add the ExtraConfig settings to **parameter_defaults** and pass the custom values. For example:

```
ExtraConfig:
    glance::config::api_config:
      glance_store/cinder_store_auth_address:
        value: "%{hiera('glance::api::authtoken::auth_url')}/v3"
      glance_store/cinder_store_user_name:
        value: <user-name>
      glance_store/cinder_store_password:
        value: "%{hiera('glance::api::authtoken::password')}"
      glance_store/cinder_store_project_name:
        value: "%{hiera('glance::api::authtoken::project_name')}"
```

18.6. IMAGE SERVICE CACHING

You can use the glance-api caching mechanism to store copies of images on your local machine and retrieve them automatically to improve scalability. If you use Image service caching, the glance-api can run on multiple hosts and does not need to retrieve the same image from backend storage multiple times. The glance-api caching mechanism does not affect any Image service operations.

You can configure Image service caching with TripleO heat templates. Set the value of **GlanceCacheEnabled** to **true**, which automatically sets **flavor** value to **keystone+cachemanagement** in the **glance-api.conf** heat template.

Use the following snippet in an environment file and include the environment file in the **openstack overcloud deploy** command when you redeploy the overcloud:

```
parameter_defaults:
    GlanceCacheEnabled: true
```

18.7. CONFIGURING THIRD PARTY STORAGE

The director include a couple of environment files to help configure third-party storage providers. This includes:

Dell EMC Storage Center

Deploys a single Dell EMC Storage Center back end for the Block Storage (cinder) service. The environment file is located at **/usr/share/openstack-tripleo-heat-templates/environments/cinder-dellsc-config.yaml**.

See the Dell Storage Center Back End Guide for full configuration information.

Dell EMC PS Series

Deploys a single Dell EMC PS Series back end for the Block Storage (cinder) service. The environment file is located at **/usr/share/openstack-tripleo-heat-templates/environments/cinder-dellps-config.yaml**.

See the Dell EMC PS Series Back End Guide for full configuration information.

NetApp Block Storage

Deploys a NetApp storage appliance as a back end for the Block Storage (cinder) service. The environment file is located at **/usr/share/openstack-tripleo-heat-templates/environments/cinder-netapp-config.yaml**.

See the NetApp Block Storage Back End Guide for full configuration information.

CHAPTER 19. SECURITY ENHANCEMENTS

The following sections provide some suggestions to harden the security of your overcloud.

19.1. MANAGING THE OVERCLOUD FIREWALL

Each of the core OpenStack Platform services contains firewall rules in their respective composable service templates. This automatically creates a default set of firewall rules for each overcloud node.

The overcloud Heat templates contain a set of parameters to help with additional firewall management:

ManageFirewall

Defines whether to automatically manage the firewall rules. Set to **true** to allow Puppet to automatically configure the firewall on each node. Set to **false** if you want to manually manage the firewall. The default is **true**.

PurgeFirewallRules

Defines whether to purge the default Linux firewall rules before configuring new ones. The default is **false**.

If **ManageFirewall** is set to **true**, you can create additional firewall rules on deployment. Set the **tripleo::firewall::firewall_rules** hieradata using a configuration hook (see Section 4.5, "Puppet: Customizing Hieradata for Roles") in an environment file for your overcloud. This hieradata is a hash containing the firewall rule names and their respective parameters as keys, all of which are optional:

port

The port associated to the rule.

dport

The destination port associated to the rule.

sport

The source port associated to the rule.

proto

The protocol associated to the rule. Defaults to **tcp**.

action

The action policy associated to the rule. Defaults to **accept**.

jump

The chain to jump to. If present, it overrides **action**.

state

An Array of states associated to the rule. Defaults to **['NEW']**.

source

The source IP address associated to the rule.

iniface

The network interface associated to the rule.

chain

The chain associated to the rule. Defaults to **INPUT**.

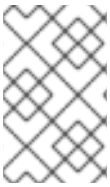
destination

The destination CIDR associated to the rule.

The following example demonstrates the syntax of the firewall rule format:

```
ExtraConfig:
  tripleo::firewall::firewall_rules:
    '300 allow custom application 1':
      port: 999
      proto: udp
      action: accept
    '301 allow custom application 2':
      port: 8081
      proto: tcp
      action: accept
```

This applies two additional firewall rules to all nodes through **ExtraConfig**.

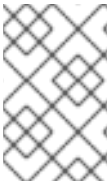


NOTE

Each rule name becomes the comment for the respective **iptables** rule. Note also each rule name starts with a three-digit prefix to help Puppet order all defined rules in the final **iptables** file. The default OpenStack Platform rules use prefixes in the 000 to 200 range.

19.2. CHANGING THE SIMPLE NETWORK MANAGEMENT PROTOCOL (SNMP) STRINGS

The director provides a default read-only SNMP configuration for your overcloud. It is advisable to change the SNMP strings to mitigate the risk of unauthorized users learning about your network devices.



NOTE

When you configure the **ExtraConfig** interface with a string parameter, you must use the following syntax to ensure that Heat and Hiera do not interpret the string as a boolean value: **"'<VALUE>'"**.

Set the following hieradata using the **ExtraConfig** hook in an environment file for your overcloud:

snmp::ro_community

IPv4 read-only SNMP community string. The default value is **public**.

snmp::ro_community6

IPv6 read-only SNMP community string. The default value is **public**.

snmp::ro_network

Network that is allowed to **RO query** the daemon. This value can be a string or an array. Default value is **127.0.0.1**.

snmp::ro_network6

Network that is allowed to **RO query** the daemon with IPv6. This value can be a string or an array. The default value is **::1/128**.

snmp::snmpd_config

Array of lines to add to the ***snmpd.conf*** file as a safety valve. The default value is **[]**. See the SNMP Configuration File web page for all available options.

For example:

-

```
parameter_defaults:
  ExtraConfig:
    snmp::ro_community: mysecurestring
    snmp::ro_community6: myv6securestring
```

This changes the read-only SNMP community string on all nodes.

19.3. CHANGING THE SSL/TLS CIPHER AND RULES FOR HAPROXY

If you enabled SSL/TLS in the overcloud (see Chapter 14, *Enabling SSL/TLS on Overcloud Public Endpoints*), you might want to harden the SSL/TLS ciphers and rules used with the HAProxy configuration. This helps avoid SSL/TLS vulnerabilities, such as the POODLE vulnerability.

Set the following hieradata using the **ExtraConfig** hook in an environment file for your overcloud:

tripleo::haproxy::ssl_cipher_suite

The cipher suite to use in HAProxy.

tripleo::haproxy::ssl_options

The SSL/TLS rules to use in HAProxy.

For example, you might aim to use the following cipher and rules:

- Cipher: **ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA384:ECDHE-ECDSA-AES256-SHA:ECDHE-RSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA256:DHE-RSA-AES256-SHA:ECDHE-ECDSA-DES-CBC3-SHA:ECDHE-RSA-DES-CBC3-SHA:EDH-RSA-DES-CBC3-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-SHA256:AES256-SHA256:AES128-SHA:AES256-SHA:DES-CBC3-SHA:!DSS**
- Rules: **no-sslv3 no-tls-tickets**

Create an environment file with the following content:

```
parameter_defaults:
  ExtraConfig:
    tripleo::haproxy::ssl_cipher_suite: ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-
CHACHA20-POLY1305:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-
SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA-
AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-
SHA256:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-
SHA384:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA384:ECDHE-ECDSA-AES256-
SHA:ECDHE-RSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-RSA-
AES256-SHA256:DHE-RSA-AES256-SHA:ECDHE-ECDSA-DES-CBC3-SHA:ECDHE-RSA-DES-
CBC3-SHA:EDH-RSA-DES-CBC3-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-
SHA256:AES256-SHA256:AES128-SHA:AES256-SHA:DES-CBC3-SHA:!DSS
    tripleo::haproxy::ssl_options: no-sslv3 no-tls-tickets
```

NOTE

The cipher collection is one continuous line.

Include this environment file with your overcloud creation.

19.4. USING THE OPEN VSWITCH FIREWALL

You can configure security groups to use the Open vSwitch (OVS) firewall driver in Red Hat OpenStack Platform director. The **NeutronOVSFirewallDriver** parameter allows you to specify which firewall driver to use:

- **iptables_hybrid** - Configures neutron to use the iptables/hybrid based implementation.
- **openvswitch** - Configures neutron to use the OVS firewall flow-based driver.

The **openvswitch** firewall driver includes higher performance and reduces the number of interfaces and bridges used to connect guests to the project network.

NOTE

The **iptables_hybrid** option is not compatible with OVS-DPDK.

Configure the **NeutronOVSFirewallDriver** parameter in the **network-environment.yaml** file:

```
NeutronOVSFirewallDriver: openvswitch
```

- **NeutronOVSFirewallDriver** : Configures the name of the firewall driver to use when implementing security groups. Possible values depend on your system configuration; some examples are: **noop**, **openvswitch**, **iptables_hybrid**. The default value of an empty string results in a supported configuration.

19.5. USING SECURE ROOT USER ACCESS

The overcloud image automatically contains hardened security for the **root** user. For example, each deployed overcloud node automatically disables direct SSH access to the **root** user. You can still access the **root** user on overcloud nodes through the following method:

1. Log into the undercloud node's **stack** user.
2. Each overcloud node has a **heat-admin** user account. This user account contains the undercloud's public SSH key, which provides SSH access without a password from the undercloud to the overcloud node. On the undercloud node, log into the chosen overcloud node through SSH using the **heat-admin** user.
3. Switch to the **root** user with **sudo -i**.

Reducing Root User Security

Some situations might require direct SSH access to the **root** user. In this case, you can reduce the SSH restrictions on the **root** user for each overcloud node.

WARNING

This method is intended for debugging purposes only. It is not recommended for use in a production environment.

The method uses the first boot configuration hook (see Section 4.1, "First Boot: Customizing First Boot Configuration"). Place the following content in an environment file:

```
resource_registry:
  OS::TripleO::NodeUserData: /usr/share/openstack-tripleo-heat-
templates/firstboot/userdata_root_password.yaml

parameter_defaults:
  NodeRootPassword: "p@55w0rd!"
```

Note the following:

- The **OS::TripleO::NodeUserData** resource refers to the a template that configures the **root** user during the first boot **cloud-init** stage.
- The **NodeRootPassword** parameter sets the password for the **root** user. Change the value of this parameter to your desired password. Note the environment file contains the password as a plain text string, which is considered a security risk.

Include this environment file with the **openstack overcloud deploy** command when creating your overcloud.

CHAPTER 20. FENCING THE CONTROLLER NODES

Fencing is the process of isolating a failed node to protect a cluster and its resources. Without fencing, a failed node can result in data corruption in a cluster.

The director uses Pacemaker to provide a highly available cluster of Controller nodes. Pacemaker uses a process called STONITH to fence failed nodes. STONITH is disabled by default and requires manual configuration so that Pacemaker can control the power management of each node in the cluster.

20.1. REVIEW THE STATE OF STONITH AND PACEMAKER

1. Log in to each node as the **heat-admin** user from the **stack** user on the director. The overcloud creation automatically copies the **stack** user's SSH key to each node's **heat-admin**.

2. Verify you have a running cluster:

```
$ sudo pcs status
Cluster name: openstackHA
Last updated: Wed Jun 24 12:40:27 2015
Last change: Wed Jun 24 11:36:18 2015
Stack: corosync
Current DC: lb-c1a2 (2) - partition with quorum
Version: 1.1.12-a14efad
3 Nodes configured
141 Resources configured
```

3. Verify STONITH is disabled:

```
$ sudo pcs property show
Cluster Properties:
cluster-infrastructure: corosync
cluster-name: openstackHA
dc-version: 1.1.12-a14efad
have-watchdog: false
stonith-enabled: false
```

20.2. ENABLE FENCING

1. Generate the **fencing.yaml** file:

```
$ openstack overcloud generate fencing --ipmi-lanplus --ipmi-level administrator --output
fencing.yaml instackenv.json
```

- Sample **fencing.yaml** file:

```
parameter_defaults:
  EnableFencing: true
  FencingConfig:
    devices:
    - agent: fence_ipmilan
      host_mac: 11:11:11:11:11:11
      params:
        ipaddr: 10.0.0.101
```

```
    lanplus: true
    login: admin
    passwd: InsertComplexPasswordHere
    pcmk_host_list: host04
    privlvl: administrator
```

2. Pass the resulting **fencing.yaml** file to the **deploy** command you previously used to deploy the overcloud. This will re-run the deployment procedure and configure fencing on the hosts:

```
openstack overcloud deploy --templates -e /usr/share/openstack-tripleo-heat-templates/environments/network-isolation.yaml -e ~/templates/network-environment.yaml -e ~/templates/storage-environment.yaml --control-scale 3 --compute-scale 3 --ceph-storage-scale 3 --control-flavor control --compute-flavor compute --ceph-storage-flavor ceph-storage --ntp-server pool.ntp.org --neutron-network-type vxlan --neutron-tunnel-types vxlan -e fencing.yaml
```

   The deployment command should complete without any error or exceptions.

3. Log in to the overcloud and verify fencing was configured for each of the controllers:

   a. Check the fencing resources are managed by Pacemaker:

```
$ source stackrc
$ nova list | grep controller
$ ssh heat-admin@<controller-x_ip>
$ sudo pcs status |grep fence
stonith-overcloud-controller-x (stonith:fence_ipmilan): Started overcloud-controller-y
```

      You should see Pacemaker is configured to use a STONITH resource for each of the controllers specified in **fencing.yaml**. The **fence-resource** process should not be configured on the same host it controls.

   b. Use **pcs** to verify the fence resource attributes:

```
$ sudo pcs stonith show <stonith-resource-controller-x>
```

      The values used by STONITH should match those defined in the **fencing.yaml**.

20.3. TEST FENCING

This procedure tests whether fencing is working as expected.

1. Trigger a fencing action for each controller in the deployment:

   a. Log in to a controller:

```
$ source stackrc
$ nova list |grep controller
$ ssh heat-admin@<controller-x_ip>
```

   b. As root, trigger fencing by using **iptables** to close all ports:

```
$ sudo -i
iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT &&
iptables -A INPUT -p tcp -m state --state NEW -m tcp --dport 22 -j ACCEPT &&
```

```
iptables -A INPUT -p tcp -m state --state NEW -m tcp --dport 5016 -j ACCEPT &&
iptables -A INPUT -p udp -m state --state NEW -m udp --dport 5016 -j ACCEPT &&
iptables -A INPUT ! -i lo -j REJECT --reject-with icmp-host-prohibited &&
iptables -A OUTPUT -p tcp --sport 22 -j ACCEPT &&
iptables -A OUTPUT -p tcp --sport 5016 -j ACCEPT &&
iptables -A OUTPUT -p udp --sport 5016 -j ACCEPT &&
iptables -A OUTPUT ! -o lo -j REJECT --reject-with icmp-host-prohibited
```

As a result, the connections should drop, and the server should be rebooted.

c. From another controller, locate the fencing event in the Pacemaker log file:

```
$ ssh heat-admin@<controller-x_ip>
$ less /var/log/cluster/corosync.log
(less): /fenc*
```

You should see that STONITH has issued a fence action against the controller, and that Pacemaker has raised an event in the log.

d. Verify the rebooted controller has returned to the cluster:

i. From the second controller, wait a few minutes and run **pcs status** to see if the fenced controller has returned to the cluster. The duration can vary depending on your configuration.

CHAPTER 21. CONFIGURING MONITORING TOOLS

Monitoring tools are an optional suite of tools that can be used for availability monitoring and centralized logging. The availability monitoring allows you to monitor the functionality of all components, while the centralized logging allows you to view all of the logs across your OpenStack environment in one central place.

For more information about configuring monitoring tools, see the dedicated Monitoring Tools Configuration Guide for full instructions.

CHAPTER 22. CONFIGURING NETWORK PLUGINS

The director includes environment files to help configure third-party network plugins:

22.1. FUJITSU CONVERGED FABRIC (C-FABRIC)

You can enable the Fujitsu Converged Fabric (C-Fabric) plugin using the environment file located at **/usr/share/openstack-tripleo-heat-templates/environments/neutron-ml2-fujitsu-cfab.yaml**.

1. Copy the environment file to your **templates** subdirectory:

```
$ cp /usr/share/openstack-tripleo-heat-templates/environments/neutron-ml2-fujitsu-cfab.yaml
/home/stack/templates/
```

2. Edit the **resource_registry** to use an absolute path:

```
resource_registry:
  OS::TripleO::Services::NeutronML2FujitsuCfab: /usr/share/openstack-tripleo-heat-
templates/puppet/services/neutron-plugin-ml2-fujitsu-cfab.yaml
```

3. Review the **parameter_defaults** in **/home/stack/templates/neutron-ml2-fujitsu-cfab.yaml**:
   - **NeutronFujitsuCfabAddress** - The telnet IP address of the C-Fabric. (string)
   - **NeutronFujitsuCfabUserName** - The C-Fabric username to use. (string)
   - **NeutronFujitsuCfabPassword** - The password of the C-Fabric user account. (string)
   - **NeutronFujitsuCfabPhysicalNetworks** - List of **<physical_network>:<vfab_id>** tuples that specify **physical_network** names and their corresponding vfab IDs. (comma_delimited_list)
   - **NeutronFujitsuCfabSharePprofile** - Determines whether to share a C-Fabric pprofile among neutron ports that use the same VLAN ID. (boolean)
   - **NeutronFujitsuCfabPprofilePrefix** - The prefix string for pprofile name. (string)
   - **NeutronFujitsuCfabSaveConfig** - Determines whether to save the configuration. (boolean)
4. To apply the template to your deployment, include the environment file in the **openstack overcloud deploy** command. For example:

```
$ openstack overcloud deploy --templates -e /home/stack/templates/neutron-ml2-fujitsu-
cfab.yaml [OTHER OPTIONS] ...
```

22.2. FUJITSU FOS SWITCH

You can enable the Fujitsu FOS Switch plugin using the environment file located at **/usr/share/openstack-tripleo-heat-templates/environments/neutron-ml2-fujitsu-fossw.yaml**.

1. Copy the environment file to your **templates** subdirectory:

```
$ cp /usr/share/openstack-tripleo-heat-templates/environments/neutron-ml2-fujitsu-fossw.yaml /home/stack/templates/
```

2. Edit the **resource_registry** to use an absolute path:

```
resource_registry:
  OS::TripleO::Services::NeutronML2FujitsuFossw: /usr/share/openstack-tripleo-heat-templates/puppet/services/neutron-plugin-ml2-fujitsu-fossw.yaml
```

3. Review the **parameter_defaults** in **/home/stack/templates/neutron-ml2-fujitsu-fossw.yaml**:

   - **NeutronFujitsuFosswIps** - The IP addresses of all FOS switches. (comma_delimited_list)
   - **NeutronFujitsuFosswUserName** - The FOS username to use. (string)
   - **NeutronFujitsuFosswPassword** - The password of the FOS user account. (string)
   - **NeutronFujitsuFosswPort** - The port number to use for the SSH connection. (number)
   - **NeutronFujitsuFosswTimeout** - The timeout period of the SSH connection. (number)
   - **NeutronFujitsuFosswUdpDestPort** - The port number of the VXLAN UDP destination on the FOS switches. (number)
   - **NeutronFujitsuFosswOvsdbVlanidRangeMin** - The minimum VLAN ID in the range that is used for binding VNI and physical port. (number)
   - **NeutronFujitsuFosswOvsdbPort** - The port number for the OVSDB server on the FOS switches. (number)

4. To apply the template to your deployment, include the environment file in the **openstack overcloud deploy** command. For example:

```
$ openstack overcloud deploy --templates -e /home/stack/templates/neutron-ml2-fujitsu-fossw.yaml [OTHER OPTIONS] ...
```

CHAPTER 23. CONFIGURING IDENTITY

The director includes parameters to help configure Identity Service (keystone) settings:

23.1. REGION NAME

By default, your overcloud's region will be named **regionOne**. You can change this by adding a **KeystoneRegion** entry your environment file. This setting cannot be changed post-deployment:

```
parameter_defaults:
  KeystoneRegion: 'SampleRegion'
```

CHAPTER 24. OTHER CONFIGURATIONS

24.1. CONFIGURING EXTERNAL LOAD BALANCING

An Overcloud uses multiple Controllers together as a high availability cluster, which ensures maximum operational performance for your OpenStack services. In addition, the cluster provides load balancing for access to the OpenStack services, which evenly distributes traffic to the Controller nodes and reduces server overload for each node. It is also possible to use an external load balancer to perform this distribution. For example, an organization might use their own hardware-based load balancer to handle traffic distribution to the Controller nodes.

For more information about configuring external load balancing, see the dedicated External Load Balancing for the Overcloud guide for full instructions.

24.2. CONFIGURING IPV6 NETWORKING

As a default, the Overcloud uses Internet Protocol version 4 (IPv4) to configure the service endpoints. However, the Overcloud also supports Internet Protocol version 6 (IPv6) endpoints, which is useful for organizations that support IPv6 infrastructure. The director includes a set of environment files to help with creating IPv6-based Overclouds.

For more information about configuring IPv6 in the Overcloud, see the dedicated IPv6 Networking for the Overcloud guide for full instructions.