Red Hat OpenStack Platform 16.2

Partner Integration

Integrating certified third-party software and hardware in a Red Hat OpenStack Platform environment

Last Updated: 2021-09-22

Red Hat OpenStack Platform 16.2 Partner Integration

Integrating certified third-party software and hardware in a Red Hat OpenStack Platform environment

OpenStack Team
rhos-docs@redhat.com

Legal Notice

Abstract

This guide provides guidelines on integrating certified third-party components into a Red Hat OpenStack Platform environment. This includes adding these components to your overcloud images and creating configuration for deployment by using director.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your input on our documentation. Tell us how we can make it better.

Using the Direct Documentation Feedback (DDF) function

Use the **Add Feedback** DDF function for direct comments on specific sentences, paragraphs, or code blocks.

1. View the documentation in the *Multi-page HTML* format.
2. Ensure that you see the **Feedback** button in the upper right corner of the document.
3. Highlight the part of text that you want to comment on.
4. Click **Add Feedback**.
5. Complete the **Add Feedback** field with your comments.
6. Optional: Add your email address so that the documentation team can contact you for clarification on your issue.
7. Click **Submit**.

CHAPTER 1. REASONS TO INTEGRATE YOUR THIRD-PARTY COMPONENETS

You can use Red Hat OpenStack Platform (RHOSP) to integrate solutions with RHOSP director. Use RHOSP director to install and manage the deployment lifecycle of a RHOSP environment. You can optimize resources, reduce deployment times, and reduce lifecycle management costs.

RHOSP director integration provides integration with existing enterprise management systems and processes. Red Hat products, such as CloudForms, are expected to have visibility into integrations with director and provide broader exposure for management of service deployment.

1.1. PARTNER INTEGRATION PREREQUISITES

You must meet several prerequisites before you can perform operations with director. The goal of partner integration is to create a shared understanding of the entire integration as a basis for Red Hat engineering, partner managers, and support resources to facilitate technology working together.

To include a third-party component with Red Hat OpenStack Platform director, you must certify the partner solution with Red Hat OpenStack Platform.

OpenStack Plug-in Certification Guides

- Red Hat OpenStack Certification Policy Guide
- Red Hat OpenStack Certification Workflow Guide

OpenStack Application Certification Guides

- Red Hat OpenStack Application Policy Guide
- Red Hat OpenStack Application Workflow Guide

OpenStack Bare Metal Certification Guides

- Red Hat OpenStack Platform Hardware Bare Metal Certification Policy Guide
- Red Hat OpenStack Platform Hardware Bare Metal Certification Workflow Guide

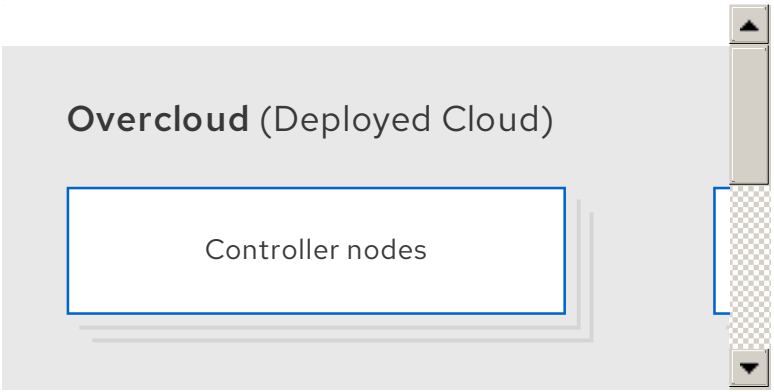
CHAPTER 2. DIRECTOR ARCHITECTURE

Red Hat OpenStack Platform director uses OpenStack APIs to configure, deploy, and manage Red Hat OpenStack Platform (RHOSP) environments. This means that integration with director requires you to integrate with these OpenStack APIs and supporting components. The benefits of these APIs are that they are well documented, undergo extensive integration testing upstream, are mature, and make understanding how director works easier for those that have a foundational knowledge of RHOSP. Director automatically inherits core OpenStack feature enhancements, security patches, and bug fixes.

Director is a toolset that you use to install and manage a complete RHOSP environment. It is based primarily on the OpenStack project TripleO, which is an abbreviation for "OpenStack-On-OpenStack". This project uses RHOSP components to install a fully operational RHOSP environment. This includes new OpenStack components that provision and control bare metal systems to use as OpenStack nodes. This provides a simple method for installing a complete RHOSP environment that is both lean and robust.

Director uses two main concepts: an undercloud and an overcloud. Director is a subset of OpenStack components that form a single-system OpenStack environment, also known as the undercloud. The undercloud acts as a management system that can create a production-level cloud for workloads to run. This production-level cloud is the overcloud. For more information about the overcloud and the undercloud, see the *Director Installation and Usage* guide.

Figure 2.1. Architecture of the undercloud and the overcloud



Director includes tools, utilities, and example templates that you can use to create an overcloud configuration. Director captures configuration data, parameters, and network topology information and uses this information in conjunction with components such as ironic, heat, and Puppet to orchestrate an overcloud installation.

2.1. CORE COMPONENTS AND OVERCLOUD

The following components are core to Red Hat OpenStack Platform director and contribute to overcloud creation:

- OpenStack Bare Metal Provisioning service (ironic)
- OpenStack Orchestration service (heat)
- Puppet
- TripleO and TripleO heat templates
- Composable services

- Containerized services and Kolla
- Ansible

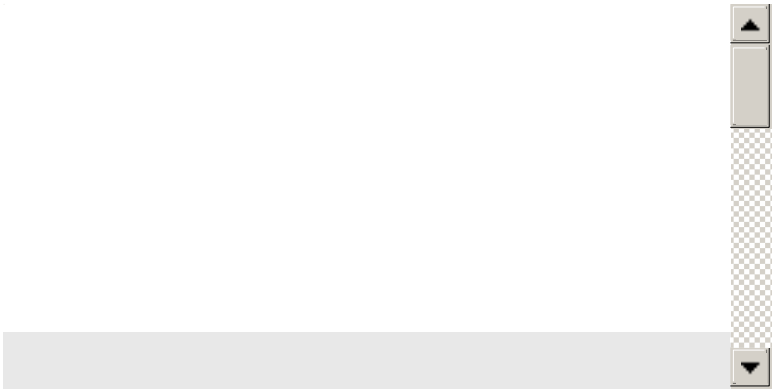
2.1.1. OpenStack Bare Metal Provisioning service (ironic)

The Bare Metal Provisioning service provides dedicated bare metal hosts to end users through self-service provisioning. Director uses Bare Metal Provisioning to manage the life-cycle of the bare metal hardware in the overcloud. Bare Metal Provisioning uses its own API to define bare metal nodes.

To provision OpenStack environments with director, you must register your nodes with Bare Metal Provisioning by using a specific driver. The main supported driver is the Intelligent Platform Management Interface (IPMI) as most hardware contains some support for IPMI power management functions. However, Bare Metal Provisioning also contains vendor specific equivalents, such as HP iLO, Cisco UCS, or Dell DRAC.

Bare Metal Provisioning controls the power management of the nodes and gathers hardware information or facts using an introspection mechanism. Director uses the information from the introspection process to match nodes to various OpenStack environment roles, such as Controller nodes, Compute nodes, and Storage nodes. For example, a discovered node with 10 disks is usually provisioned as a Storage node.

Figure 2.2. Use the Bare Metal Provisioning service controls the power management of the nodes

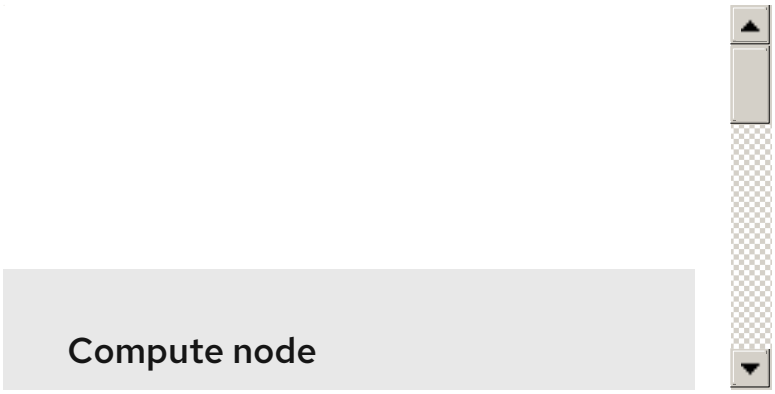


If you want to have director support for your hardware, you must have driver coverage in the Bare Metal Provisioning service.

2.1.2. Heat

Heat is an application stack orchestration engine. You can use heat to define elements for an application before you deploy it to a cloud. Create a stack template that includes a number of infrastructure resources, for example, instances, networks, storage volumes, and elastic IPs, with a set of parameters for configuration. Use heat to create these resources based on a given dependency chain, monitor the resources for availability, and scale if necessary. You can use these templates to make application stacks portable and to achieve repeatable results.

Figure 2.3. Use the heat service to define elements for an application before you deploy it to a cloud



Director uses the native OpenStack heat APIs to provision and manage the resources associated with overcloud deployment. This includes precise details such as defining the number of nodes to provision per node role, the software components to configure for each node, and the order in which director configures these components and node types. Director also uses heat to troubleshoot a deployment and make changes post-deployment.

The following example is a snippet from a heat template that defines parameters of a Controller node:

```
NeutronExternalNetworkBridge:
  description: Name of bridge used for external network traffic.
  type: string
  default: 'br-ex'
NeutronBridgeMappings:
  description: >
    The OVS logical->physical bridge mappings to use. See the Neutron
    documentation for details. Defaults to mapping br-ex - the external
    bridge on hosts - to a physical name 'datacentre' which can be used
    to create provider networks (and we use this for the default floating
    network) - if changing this either use different post-install network
    scripts or be sure to keep 'datacentre' as a mapping network name.
  type: string
  default: "datacentre:br-ex"
```

Heat consumes templates included with the director to facilitate the creation of an overcloud, which includes calling ironic to power the nodes. You can use the standard heat tools to view the resources and statuses of an in-progress overcloud. For example, you can use the heat tools to display the overcloud as a nested application stack. Use the syntax of heat templates to declare and create production OpenStack clouds. Because every partner integration use case requires heat templates, you must have some prior understanding and proficiency for partner integration.

2.1.3. Puppet

Puppet is a configuration management and enforcement tool you can use to describe and maintain the end state of a machine. You define this end state in a Puppet manifest. Puppet supports two models:

- A standalone mode in which you run instructions in the form of manifests locally
- A server mode in which Puppet retrieves its manifests from a central server, called a Puppet Master

You can make changes in two ways:

- Upload new manifests to a node and execute them locally.
- Make modifications in the client/server model on the Puppet Master.

Director uses Puppet in the following areas:

- On the undercloud host locally to install and configure packages according to the configuration in the **undercloud.conf** file.
- By injecting the **openstack-puppet-modules** package into the base overcloud image, the Puppet modules are ready for post-deployment configuration. By default, you create an image that contains all OpenStack services for each node.
- Providing additional Puppet manifests and heat parameters to the nodes and applying the configuration after overcloud deployment. This includes the services to enable and start the configuration depending on the node type.
- Providing Puppet hieradata to the nodes. The Puppet modules and manifests are free from site or node-specific parameters to keep the manifests consistent. The hieradata acts as a form of parameterized values that you can push to a Puppet module and reference in other areas. For example, to reference the MySQL password inside of a manifest, save this information as hieradata and reference it within the manifest.
  To view the hieradata, enter the following command:

  ```
  [root@localhost ~]# grep mysql_root_password hieradata.yaml # View the data in the hieradata file
  openstack::controller::mysql_root_password: ‘redhat123'
  ```

  To reference the hieradata in the Puppet manifest, enter the following command:s

  ```
  [root@localhost ~]# grep mysql_root_password example.pp # Now referenced in the Puppet manifest
  mysql_root_password  => hiera(‘openstack::controller::mysql_root_password')
  ```

Partner-integrated services that need package installation and service enablement can create Puppet modules to meet their requirements. For more information about obtaining current OpenStack Puppet modules and examples, see Section 4.2, “Obtaining OpenStack Puppet modules”.

2.1.4. TripleO and TripleO heat templates

Director is based on the upstream TripleO project. This project combines a set of OpenStack services with the following goals:

- Store overcloud images by using the Image service (glance)
- Orchestrate the overcloud by using the Orchestration service (heat)
- Provision bare metal machines by using the Bare Metal Provisioning (ironic) and Compute (nova) services

TripleO also includes a heat template collection that defines a Red Hat-supported overcloud environment. Director, using heat, reads this template collection and orchestrates the overcloud stack.

2.1.5. Composable services

Each aspect of Red Hat OpenStack Platform is broken into a composable service. This means that you can define different roles that use different combinations of services. For example, you can move the networking agents from the default Controller node to a standalone Networker node.

For more information about the composable service architecture, see Chapter 6, *Composable services*.

2.1.6. Containerized services and Kolla

Each of the main Red Hat OpenStack Platform (RHOSP) services run in containers. This provides a method to keep each service within its own isolated namespace separated from the host. This has the following effects:

- During deployment, RHOSP pulls and runs container images from the Red Hat Customer Portal.
- The **podman** command operates management functions, like starting and stopping services.
- To upgrade containers, you must pull new container images and replace the existing containers with newer versions.

Red Hat OpenStack Platform uses a set of containers built and managed with the **Kolla** toolset.

2.1.7. Ansible

Red Hat OpenStack Platform uses Ansible to drive certain functions in relation to composable service upgrades. This includes functions such as starting and stopping services and performing database upgrades. These upgrade tasks are defined within composable service templates.

CHAPTER 3. WORKING WITH OVERCLOUD IMAGES

Red Hat OpenStack Platform (RHOSP) director provides images for the overcloud. The QCOW image in this collection contains a base set of software components that integrate to form various overcloud roles, such as Compute, Controller, and storage nodes. In some situations, you might aim to modify certain aspects of the overcloud image to suit your needs, such as installing additional components to nodes.

You can use the **virt-customize** tool to modify an existing overcloud image to augment an existing Controller node. For example, use the following procedures to install additional **ml2** plugins, Cinder backends, or monitoring agents that do not ship with the initial image.

IMPORTANT

If you modify the overcloud image to include third-party software and report an issue, Red Hat might request that you reproduce the issue with an unmodified image in accordance with our general third-party support policy: https://access.redhat.com/articles/1067.

3.1. OBTAINING THE OVERCLOUD IMAGES

Director requires several disk images to provision overcloud nodes:

- **An introspection kernel and ramdisk** - For bare metal system introspection over PXE boot.
- **A deployment kernel and ramdisk** - For system provisioning and deployment.
- **An overcloud kernel, ramdisk, and full image** - A base overcloud system that director writes to the hard disk of the node.

Procedure

1. To obtain these images, install the **rhosp-director-images** and **rhosp-director-images-ipa** packages:

```
$ sudo dnf install rhosp-director-images rhosp-director-images-ipa
```

2. Extract the archives to the **images** directory on the **stack** user home, **/home/stack/images**:

```
$ cd ~/images
$ for i in /usr/share/rhosp-director-images/overcloud-full-latest-16.2.tar /usr/share/rhosp-director-images/ironic-python-agent-latest-16.2.tar; do tar -xvf $i; done
```

3.2. INITRD: MODIFYING THE INITIAL RAMDISKS

Some situations might require that you modify the initial ramdisk. For example, you might require that a certain driver is available when you boot the nodes during the introspection or provisioning processes. In the context of the overcloud, this includes one of the following ramdisks:

- The introspection ramdisk - **ironic-python-agent.initramfs**
- The provisioning ramdisk - **overcloud-full.initrd**

This procedure adds an additional RPM package to the **ironic-python-agent.initramfs** ramdisk as an example.

Prerequisites

- You have installed the pax utility:

```
$ sudo dnf install -y spax
```

Procedure

1. Log in as the **root** user and create a temporary directory for the ramdisk:

```
# mkdir ~/ipa-tmp
# cd ~/ipa-tmp
```

2. Use the **skipcpio** and **cpio** commands to extract the ramdisk to the temporary directory:

```
# /usr/lib/dracut/skipcpio ~/images/ironic-python-agent.initramfs | zcat | cpio -ivd | pax -r
```

3. Install an RPM package to the extracted contents:

```
# rpm2cpio ~/RPMs/python-proliantutils-2.1.7-1.el7ost.noarch.rpm | pax -r
```

4. Recreate the new ramdisk:

```
# find . 2>/dev/null | cpio --quiet -c -o | gzip -8  > /home/stack/images/ironic-python-agent.initramfs
# chown stack: /home/stack/images/ironic-python-agent.initramfs
```

5. Verify that the new package now exists in the ramdisk:

```
# lsinitrd /home/stack/images/ironic-python-agent.initramfs | grep proliant
```

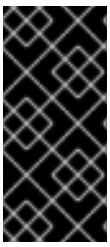
3.3. QCOW: INSTALLING VIRT-CUSTOMIZE TO DIRECTOR

The **libguestfs-tools** package contains the **virt-customize** tool.

Procedure

- Install the **libguestfs-tools** from the **rhel-8-for-x86_64-appstream-eus-rpms** repository:

```
$ sudo dnf install libguestfs-tools
```



IMPORTANT

If you install the **libguestfs-tools** package on the undercloud, disable **iscsid.socket** to avoid port conflicts with the **tripleo_iscsid** service on the undercloud:

```
$ sudo systemctl disable --now iscsid.socket
```

3.4. QCOW: INSPECTING THE OVERCLOUD IMAGE

Before you can review the contents of the **overcloud-full.qcow2** image, you must create a virtual machine that uses this image.

Procedure

1. To create a virtual machine instance that uses the **overcloud-full.qcow2** image, use the **guestmount** command:

```
$ mkdir ~/overcloud-full
$ guestmount -a overcloud-full.qcow2 -i --ro ~/overcloud-full
```

   You can review the contents of the QCOW2 image in ~/**overcloud-full**.

2. Alternatively, you can use virt-manager to create a virtual machine with the following boot options:

   - **Kernel path:** /overcloud-full.vmlinuz
   - **initrd path:** /overcloud-full.initrd
   - **Kernel arguments:** root=/dev/sda

3.5. QCOW: SETTING THE ROOT PASSWORD

Set the root password to provide administrator-level access for your nodes through the console.

Procedure

- Set the password for the **root** user on image:

```
$ virt-customize --selinux-relabel -a overcloud-full.qcow2 --root-password password:test
[  0.0] Examining the guest ...
[ 18.0] Setting a random seed
[ 18.0] Setting passwords
[ 19.0] Finishing off
```

3.6. QCOW: REGISTERING THE IMAGE

Register your overcloud image to the Red Hat Content Delivery Network.

Procedure

1. Register your image temporarily to enable Red Hat repositories relevant to your customizations:

```
$ virt-customize --selinux-relabel -a overcloud-full.qcow2 --run-command 'subscription-
manager register --username=[username] --password=[password]'
[  0.0] Examining the guest ...
[ 10.0] Setting a random seed
[ 10.0] Running: subscription-manager register --username=[username] --password=
[password]
[ 24.0] Finishing off
```

2. Replace the **[username]** and **[password]** with your Red Hat customer account details. This runs the following command on the image:

```
subscription-manager register --username=[username] --password=[password]
```

3.7. QCOW: ATTACHING A SUBSCRIPTION AND ENABLING RED HAT REPOSITORIES

Procedure

1. Find a list of pool ID from your account subscriptions:

```
$ sudo subscription-manager list
```

2. Choose a subscription pool ID and attach it to the image:

```
$ virt-customize --selinux-relabel -a overcloud-full.qcow2 --run-command 'subscription-
manager attach --pool [subscription-pool]'
[  0.0] Examining the guest ...
[ 12.0] Setting a random seed
[ 12.0] Running: subscription-manager attach --pool [subscription-pool]
[ 52.0] Finishing off
```

3. Replace the **[subscription-pool]** with your chosen subscription pool ID:

```
subscription-manager attach --pool [subscription-pool]
```

This adds the pool to the image so that you can enable the repositories.

4. Enable the Red Hat repositories:

```
$ subscription-manager repos --enable=[repo-id]
```

3.8. QCOW: COPYING A CUSTOM REPOSITORY FILE

Adding third-party software to the image requires additional repositories. The following is an example repo file that contains configuration to use the OpenDaylight repository content.

Procedure

1. List the contents of the **opendaylight.repo** file:

```
$ cat opendaylight.repo

[opendaylight]
name=OpenDaylight Repository
baseurl=https://nexus.opendaylight.org/content/repositories/opendaylight-yum-epel-8-
x86_64/
gpgcheck=0
```

2. Copy the repository file on to the image:

```
$ virt-customize --selinux-relabel -a overcloud-full.qcow2 --upload
opendaylight.repo:/etc/yum.repos.d/
[  0.0] Examining the guest ...
[ 12.0] Setting a random seed
[ 12.0] Copying: opendaylight.repo to /etc/yum.repos.d/
[ 13.0] Finishing off
```

The **--upload** option copies the repository file to **/etc/yum.repos.d/** on the overcloud image.

IMPORTANT

Red Hat does not offer support for software from non-certified vendors. Check with your Red Hat support representative that the software you want to install is supported.

3.9. QCOW: INSTALLING RPMS

Procedure

- Use the **virt-customize** command to install packages to the image:

```
$ virt-customize --selinux-relabel -a overcloud-full.qcow2 --install opendaylight
[  0.0] Examining the guest ...
[ 11.0] Setting a random seed
[ 11.0] Installing packages: opendaylight
[ 91.0] Finishing off
```

Use the **--install** option to specify a package to install.

3.10. QCOW: CLEANING THE SUBSCRIPTION POOL

Procedure

- After you install the necessary packages to customize the image, remove the subscription pools and unregister the image:

```
$ virt-customize --selinux-relabel -a overcloud-full.qcow2 --run-command 'subscription-
manager remove --all'
[  0.0] Examining the guest ...
[ 12.0] Setting a random seed
[ 12.0] Running: subscription-manager remove --all
[ 18.0] Finishing off
```

3.11. QCOW: UNREGISTERING THE IMAGE

Procedure

- Unregister the image so that the overcloud deployment process can deploy the image to your nodes and register each of them individually:

```
$ virt-customize --selinux-relabel -a overcloud-full.qcow2 --run-command 'subscription-
manager unregister'
[  0.0] Examining the guest ...
```

```
[ 11.0] Setting a random seed
[ 11.0] Running: subscription-manager unregister
[ 17.0] Finishing off
```

3.12. QCOW: RESET THE MACHINE ID

Procedure

- Reset the machine ID for the image so that machines that use this image do not use duplicate machine IDs:

```
$ virt-sysprep --operation machine-id -a overcloud-full.qcow2
```

3.13. UPLOADING THE IMAGES TO DIRECTOR

After you modify the image, you need to upload it to director.

Procedure

1. Source the **stackrc** file so that you can access director from the command line:

```
$ source stackrc
```

2. Upload the default director images to use for deploying the overcloud:

```
$ openstack overcloud image upload --image-path /home/stack/images/
```

This uploads the following images into the director:

- bm-deploy-kernel
- bm-deploy-ramdisk
- overcloud-full
- overcloud-full-initrd
- overcloud-full-vmlinuz
  The script also installs the introspection images on the directors PXE server.

3. View a list of the images in the CLI:

```
$ openstack image list
+--------------------------------------+------------------------+
| ID                                   | Name                   |
+--------------------------------------+------------------------+
| 765a46af-4417-4592-91e5-a300ead3faf6 | bm-deploy-ramdisk      |
| 09b40e3d-0382-4925-a356-3a4b4f36b514 | bm-deploy-kernel       |
| ef793cd0-e65c-456a-a675-63cd57610bd5 | overcloud-full         |
| 9a51a6cb-4670-40de-b64b-b70f4dd44152 | overcloud-full-initrd  |
| 4f7e33f4-d617-47c1-b36f-cbe90f132e5d | overcloud-full-vmlinuz |
+--------------------------------------+------------------------+
```

This list does not show the introspection PXE images (agent.*). Director copies these files to **/httpboot**.

```
[stack@host1 ~]$ ls /httpboot -l
total 151636
-rw-r--r--. 1 ironic ironic       269 Sep 19 02:43 boot.ipxe
-rw-r--r--. 1 root   root         252 Sep 10 15:35 inspector.ipxe
-rwxr-xr-x. 1 root   root     5027584 Sep 10 16:32 agent.kernel
-rw-r--r--. 1 root   root   150230861 Sep 10 16:32 agent.ramdisk
drwxr-xr-x. 2 ironic ironic       4096 Sep 19 02:45 pxelinux.cfg
```

CHAPTER 4. CONFIGURING ADDITIONS TO THE OPENSTACK PUPPET MODULES

This chapter explores how to provide additions to the OpenStack Puppet modules. This includes some basic guidelines on developing Puppet modules.

4.1. PUPPET SYNTAX AND MODULE STRUCTURE

The following section provides a few basics to help you understand Puppet syntax and the structure of a Puppet module.

4.1.1. Anatomy of a Puppet module

Before you contribute to the OpenStack modules, you must understand the components that create a Puppet module.

Manifests

Manifests are files that contain code to define a set of resources and their attributes. A resource is any configurable part of a system. Examples of resources include packages, services, files, users, and groups, SELinux configuration, SSH key authentication, cron jobs, and more. A manifest defines each required resource by using a set of key-value pairs for their attributes.

```
package { 'httpd':
  ensure => installed,
}
```

For example, this declaration checks if the **httpd** package is installed. If not, the manifest executes **dnf** and installs it. Manifests are located in the manifest directory of a module. Puppet modules also use a test directory for test manifests. These manifests are used to test certain classes that are in your official manifests.

Classes

Classes unify multiple resources in a manifest. For example, if you install and configure a HTTP server, you might create a class with three resources: one to install the HTTP server packages, one to configure the HTTP server, and one to start or enable the server. You can also refer to classes from other modules, which applies their configuration. For example, if you want to configure an application that also required a webserver, you can refer to the previously mentioned class for the HTTP server.

Static Files

Modules can contain static files that Puppet can copy to certain locations on your system. Define locations, and other attributes such as permissions, by using file resource declarations in manifests. Static files are located in the files directory of a module.

Templates

Sometimes configuration files require custom content. In this situation, users create a template instead of a static file. Like static files, templates are defined in manifests and copied to locations on a system. The difference is that templates allow Ruby expressions to define customized content and variable input. For example, if you want to configure httpd with a customizable port then the template for the configuration file includes:

```
Listen <%= @httpd_port %>
```

The **httpd_port** variable in this case is defined in the manifest that references this template.

Templates are located in the templates directory of a module.

Plugins

Use plugins for aspects that extend beyond the core functionality of Puppet. For example, you can use plugins to define custom facts, custom resources, or new functions. For example, a database administrator might need a resource type for PostgreSQL databases. This can help the database administrator populate PostgreSQL with a set of new databases after they install PostgreSQL. As a result, the database administrator must create only a Puppet manifest that ensures PostgreSQL installs and the databases are created afterwards.
Plugins are located in the lib directory of a module. This includes a set of subdirectories depending on the plugin type:

- **/lib/facter** - Location for custom facts.
- **/lib/puppet/type** - Location for custom resource type definitions, which outline the key-value pairs for attributes.
- **/lib/puppet/provider** - Location for custom resource providers, which are used in conjunction with resource type definitions to control resources.
- **/lib/puppet/parser/functions** - Location for custom functions.

4.1.2. Installing a service

Some software requires package installations. This is one function that a Puppet module can perform. This requires a resource definition that defines configurations for a certain package.

For example, to install the **httpd** package through the **mymodule** module, add the following content to a Puppet manifest in the **mymodule** module:

```
class mymodule::httpd {
  package { 'httpd':
    ensure => installed,
  }
}
```

This code defines a subclass of **mymodule** called **httpd**, then defines a package resource declaration for the **httpd** package. The **ensure => installed** attribute tells Puppet to check if the package is installed. If it is not installed, Puppet executes **dnf** to install it.

4.1.3. Starting and enabling a service

After you install a package, you might want to start the service. Use another resource declaration called **service**. Edit the manifest with the following content:

```
class mymodule::httpd {
  package { 'httpd':
    ensure => installed,
  }
  service { 'httpd':
    ensure => running,
```

```
    enable => true,
    require => Package["httpd"],
  }
}
```

Result:

- The **ensure => running** attribute checks if the service is running. If not, Puppet enables it.
- The **enable => true** attribute sets the service to run when the system boots.
- The **require => Package["httpd"]** attribute defines an ordering relationship between one resource declaration and another. In this case, it ensures that the **httpd** service starts after the **httpd** package installs. This creates a dependency between the service and its respective package.

4.1.4. Configuring a service

The HTTP server provides some default configuration in **/etc/httpd/conf/httpd.conf**, which provides a web host on port 80. However, you can add extra configuration to provide an additional web host on a user-specified port.

Procedure

1. You must use a template file to store the HTTP configuration file because the user-defined port requires variable input. In the module templates directory, add a file called **myserver.conf.erb** with the following contents:

```
Listen <%= @httpd_port %>
NameVirtualHost *:<%= @httpd_port %>
<VirtualHost *:<%= @httpd_port %>>
  DocumentRoot /var/www/myserver/
  ServerName *:<%= @fqdn %>>
  <Directory "/var/www/myserver/">
    Options All Indexes FollowSymLinks
    Order allow,deny
    Allow from all
  </Directory>
</VirtualHost>
```

   This template follows the standard syntax for Apache web server configuration. The only difference is the inclusion of Ruby escape characters to inject variables from the module. For example, **httpd_port**, which you use to specify the web server port.

   The inclusion of **fqdn** is a variable that stores the fully qualified domain name of the system. This is known as a **system fact**. System facts are collected from each system before generating each Puppet catalog of a system. Puppet uses the **facter** command to gather these system facts and you can also run **facter** to view a list of these facts.

2. Save **myserver.conf.erb**.

3. Add the resource to the Puppet manifest of the module:

```
class mymodule::httpd {
  package { 'httpd':
```

```
    ensure => installed,
  }
  service { 'httpd':
    ensure => running,
    enable => true,
    require => Package["httpd"],
  }
  file {'/etc/httpd/conf.d/myserver.conf':
  notify => Service["httpd"],
    ensure => file,
    require => Package["httpd"],
    content => template("mymodule/myserver.conf.erb"),
  }
  file { "/var/www/myserver":
    ensure => "directory",
  }
}
```

Result:

- You add a file resource declaration for the server configuration file, **(/etc/httpd/conf.d/myserver.conf**. The content for this file is the **myserver.conf.erb** template that you created.
- You check the **httpd** package is installed before you add this file.
- You add a second file resource declaration that creates a directory, **/var/www/myserver**, for your web server.
- You add a relationship between the configuration file and the **httpd** service using the **notify => Service["httpd"]** attribute. This checks your configuration file for any changes. If the file has changed, Puppet restarts the service.

4.2. OBTAINING OPENSTACK PUPPET MODULES

The Red Hat OpenStack Platform uses the official OpenStack Puppet modules. To obtain OpenStack Puppet modules, see the **openstack** group on **Github**.

Procedure

1. In your browser, go to https://github.com/openstack.
2. In the filters section, search for **Puppet**. All Puppet modules use the prefix **puppet-**.
3. Clone the Puppet module that you want. For example, the official OpenStack Block Storage (cinder) module:

```
$ git clone https://github.com/openstack/puppet-cinder.git
```

4.3. EXAMPLE CONFIGURATION OF A PUPPET MODULE

The OpenStack modules primarily aim to configure the core service. Most modules also contain additional manifests to configure additional services, sometimes known as **backends**, **agents**, or **plugins**. For example, the **cinder** module contains a directory called **backends**, which contains configuration

options for different storage devices including NFS, iSCSI, Red Hat Ceph Storage, and others.

For example, the **manifests/backends/nfs.pp** file contains the following configuration:

```
define cinder::backend::nfs (
  $volume_backend_name  = $name,
  $nfs_servers          = [],
  $nfs_mount_options    = undef,
  $nfs_disk_util        = undef,
  $nfs_sparsed_volumes  = undef,
  $nfs_mount_point_base = undef,
  $nfs_shares_config    = '/etc/cinder/shares.conf',
  $nfs_used_ratio       = '0.95',
  $nfs_oversub_ratio    = '1.0',
  $extra_options        = {},
) {

  file {$nfs_shares_config:
    content => join($nfs_servers, "\n"),
    require => Package['cinder'],
    notify  => Service['cinder-volume']
  }

  cinder_config {
    "${name}/volume_backend_name":  value => $volume_backend_name;
    "${name}/volume_driver":        value =>
      'cinder.volume.drivers.nfs.NfsDriver';
    "${name}/nfs_shares_config":    value => $nfs_shares_config;
    "${name}/nfs_mount_options":    value => $nfs_mount_options;
    "${name}/nfs_disk_util":        value => $nfs_disk_util;
    "${name}/nfs_sparsed_volumes":  value => $nfs_sparsed_volumes;
    "${name}/nfs_mount_point_base": value => $nfs_mount_point_base;
    "${name}/nfs_used_ratio":       value => $nfs_used_ratio;
    "${name}/nfs_oversub_ratio":    value => $nfs_oversub_ratio;
  }

  create_resources('cinder_config', $extra_options)

}
```

Result:

- The **define** statement creates a defined type called **cinder::backend::nfs**. A defined type is similar to a class; the main difference is Puppet evaluates a defined type multiple times. For example, you might require multiple NFS back ends and as such the configuration requires multiple evaluations for each NFS share.
- The next few lines define the parameters in this configuration and their default values. The default values are overwritten if the user passes new values to the **cinder::backend::nfs** defined type.
- The **file** function is a resource declaration that calls for the creation of a file. This file contains a list of the NFS shares and the name for this file is defined in the parameters **$nfs_shares_config = '/etc/cinder/shares.conf**. Note the additional attributes:
  - The **content** attribute creates a list by using the **$nfs_servers** parameter.

- The **require** attribute ensures that the **cinder** package is installed.
- The **notify** attribute tells the **cinder-volume** service to reset.

- The **cinder_config** function is a resource declaration that uses a plugin from the **lib/puppet/** directory in the module. This plugin adds configuration to the **/etc/cinder/cinder.conf** file. Each line in this resource adds a configuration options to the relevant section in the **cinder.conf** file. For example, if the **$name** parameter is **mynfs**, then the following attributes:

```
"${name}/volume_backend_name":  value => $volume_backend_name;
"${name}/volume_driver":        value =>
  'cinder.volume.drivers.nfs.NfsDriver';
"${name}/nfs_shares_config":    value => $nfs_shares_config;
```

  Save the following snippet to the **cinder.conf** file:

```
[mynfs]
volume_backend_name=mynfs
volume_driver=cinder.volume.drivers.nfs.NfsDriver
nfs_shares_config=/etc/cinder/shares.conf
```

- The **create_resources** function converts a hash into a set of resources. In this case, the manifest converts the **$extra_options** hash to a set of additional configuration options for the backend. This provides a flexible method to add further configuration options that are not included in the core parameters of the manifest.

This shows the importance of including a manifest to configure the OpenStack driver of your hardware. The manifest provides a method for director to include configuration options that are relevant to your hardware. This acts as a main integration point for director to configure your overcloud to use your hardware.

4.4. EXAMPLE OF ADDING HIERA DATA TO A PUPPET CONFIGURATION

Puppet contains a tool called **hiera**, which acts as a key value system that provides node-specific configuration. These keys and their values are usually stored in files located in **/etc/puppet/hieradata**. The **/etc/puppet/hiera.yaml** file defines the order that Puppet reads the files in the **hieradata** directory.

During overcloud configuration, Puppet uses hiera data to overwrite the default values for certain Puppet classes. For example, the default NFS mount options for **cinder::backend::nfs** in **puppet-cinder** are undefined:

```
$nfs_mount_options    = undef,
```

However, you can create your own manifest that calls the **cinder::backend::nfs** defined type and replace this option with hiera data:

```
cinder::backend::nfs { $cinder_nfs_backend:
  nfs_mount_options   => hiera('cinder_nfs_mount_options'),
}
```

This means the **nfs_mount_options** parameter takes uses hiera data value from the **cinder_nfs_mount_options** key:

```
cinder_nfs_mount_options: rsize=8192,wsize=8192
```

Alternatively, you can use the hiera data to overwrite **cinder::backend::nfs::nfs_mount_options** parameter directly so that it applies to all evaluations of the NFS configuration:

```
cinder::backend::nfs::nfs_mount_options: rsize=8192,wsize=8192
```

The above hiera data overwrites this parameter on each evaluation of **cinder::backend::nfs**.

CHAPTER 5. ORCHESTRATION

Director uses Heat Orchestration Templates (HOT) as the template format for the 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 OpenStack Orchestration (heat) creates, and the configuration of the resources. Resources are objects in Red Hat OpenStack Platform (RHOSP) and can include compute resources, network configuration, security groups, scaling rules, and custom resources.

NOTE

For RHOSP to use the heat template file as a custom template resource, the file extension must be either .yaml or .template.

5.1. LEARNING HEAT TEMPLATE BASICS

5.1.1. Understanding heat templates

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

Use the **resources** section to define the resources, such as compute instances, networks, and storage volumes, that you can create when you deploy a stack using this template. Red Hat OpenStack Platform (RHOSP) contains a set of core resources that span across all components. These are the specific objects to create and configure as part of a stack. RHOSP contains a set of core resources that span across all components. These are defined in the **resources** section of a template.

outputs

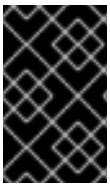
Use the **outputs** section to declare the output parameters that your cloud users can access after the stack is created. Your cloud users can use these parameters to request details about the stack, such as the IP addresses of deployed instances, or URLs of web applications deployed as part of the stack.

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 that the cloud user specifies. The stack can return the value of **instance_name**, which is called **My Cirros Instance**.



IMPORTANT

A heat template also requires the **heat_template_version** parameter, which defines the syntax version to use and the functions available. For more information, see the Official Heat Documentation.

5.1.2. Understanding environment files

An environment file is a special type of template that you can use to customize your heat templates. You can include environment files in the deployment command, in addition to the core heat templates. An environment file contains three main sections:

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.

parameters

These are common settings that you apply to the parameters of the top-level template. For example, if you have a template that deploys nested stacks, such as resource registry mappings, the parameters apply only to the top-level template and not to templates for the nested resources.

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.

IMPORTANT

Use **parameter_defaults** instead of **parameters** when you create custom environment files for your overcloud, so that your parameters apply to all stack templates for the overcloud.

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
```

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** file 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.

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

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

NOTE

For RHOSP to use the heat template file as a custom template resource, the file extension must be either .yaml or .template.

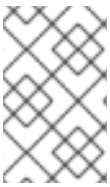
5.2. OBTAINING THE DEFAULT DIRECTOR TEMPLATES

Director uses an advanced heat template collection to create an overcloud. This collection is available from the **openstack** group on **Github** in the **openstack-tripleo-heat-templates** repository.

Procedure

- To obtain a clone of this template collection, enter the following command:

```
$ git clone https://github.com/openstack/tripleo-heat-templates.git
```



NOTE

The Red Hat-specific version of this template collection is available from the **openstack-tripleo-heat-template** package, which installs the collection to **/usr/share/openstack-tripleo-heat-templates**.

The main files and directories in this template collection are:

overcloud.j2.yaml

This is the main template file that director uses 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 that director uses to create the overcloud environment. It provides a

set of configurations for Puppet modules stored on the overcloud image. After director writes the overcloud image to each node, heat starts the Puppet configuration for each node by 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

This file contains the definitions of the roles in an overcloud and maps services to each role.

network_data.yaml

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

plan-environment.yaml

This file contains the definitions of 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

This file contains a mapping of environment files for an overcloud plan.

deployment

This directory contains heat templates. The **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.

environments

This directory contains additional heat environment files that you can use for your overcloud creation. These environment files enable extra functions for your resulting Red Hat OpenStack Platform (RHOSP) environment. For example, the directory contains an environment file to enable Cinder NetApp backend storage (**cinder-netapp-config.yaml**).

network

This directory contains a set of heat templates that you can use to create isolated networks and ports.

puppet

This directory contains templates that control Puppet configuration. The **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

This directory contains legacy heat templates for all service configuration. The templates in the **deployment** directory replace most of the templates in the **puppet/services** directory.

extraconfig

This directory contains templates that you can use to enable extra functionality.

firstboot

This directory contains example **first_boot** scripts that director uses when initially creating the nodes.

IMPORTANT

Previous versions of this guide contained reference material for using configuration hooks to integrate services. The recommended method for partners to integrate services is to now use the composable service framework. For more information, see Chapter 6, *Composable services*.

CHAPTER 6. COMPOSABLE SERVICES

Red Hat OpenStack Platform (RHOSP) includes the ability to define custom roles and compose service combinations on roles. For more information, see Composable Services and Custom Roles in the *Advanced Overcloud Customization* guide. As part of the integration, you can define your own custom services and include them on chosen roles.

6.1. EXAMINING COMPOSABLE SERVICE ARCHITECTURE

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

- **deployment** contains the templates for key OpenStack 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 that 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:

```
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:

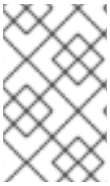
```
{{role.name}}Services:
  description: A list of service resources (configured in the heat
               resource_registry) which represent nested stacks
               for each service that should get installed on the {{role.name}} role.
  type: comma_delimited_list
  default: {{role.ServicesDefault|default([])}}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

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.

6.2. CREATING A USER-DEFINED COMPOSABLE SERVICE

This example examines how to create a user-defined composable service and focuses on implementing a message of the day (**motd**) service. In this example, the overcloud image contains a custom **motd** Puppet module loaded either through a configuration hook or through modifying the overcloud images. For more information, see Chapter 3, *Working with overcloud images* .

When you create your own service, you must define the following items in the heat template of your service:

parameters

- **ServiceData** - A map of service specific data. Use an empty hash (**{}**) as the **default** value because values from the heat template override this parameter.
- **ServiceNetMap** - A map of services to networks. Use an empty hash (**{}**) as the **default** value because values from the heat template override this parameter.
- **EndpointMap** - A list of OpenStack service endpoints to protocols. Use an empty hash (**{}**) as the **default** value because values from the heat template override this parameter
- **DefaultPasswords** - A list of default passwords. Use an empty hash (**{}**) as the **default** value because values from the heat template override this parameter.
- **RoleName** - Name of the role that this service is deployed. Use an empty string (**''**) as the **default** value because values from the heat template override this parameter.
- **RoleParameters** - Parameters specific to a role. These parameters are defined with the **<RoleName>Parameters** parameter, replace **<RoleName>** with the name of the role. Use an empty hash (**{}**) as the **default** value because values from the heat template override this parameter

Define any additional parameters that your service requires.

outputs

The following output parameters define the service configuration on the host. For more information, see Appendix A, *Composable service parameters*.

The following is an example heat template (**service.yaml**) for the **motd** service:

```
heat_template_version: 2016-04-08

description: >
  Message of the day service configured with Puppet

parameters:
  ServiceNetMap:
    default: {}
    type: json
  DefaultPasswords:
    default: {}
    type: json
  EndpointMap:
    default: {}
    type: json
  MotdMessage: 1
    default: |
      Welcome to my Red Hat OpenStack Platform environment!

    type: string
    description: The message to include in the motd

outputs:
  role_data:
    description: Motd role using composable services.
    value:
```

```
  service_name: motd
  config_settings: 2
    motd::content: {get_param: MotdMessage}
  step_config: | 3
    if hiera('step') >= 2 {
      include ::motd
    }
```

1. The template includes a **MotdMessage** parameter that defines the message of the day. The parameter includes a default message but you can override it by using the same parameter in a custom environment file.

2. The **outputs** section defines some service hieradata in **config_settings**. The **motd::content** hieradata stores the content from the **MotdMessage** parameter. The **motd** Puppet class eventually reads this hieradata and passes the user-defined message to the **/etc/motd** file.

3. The **outputs** section includes a Puppet manifest snippet in **step_config**. The snippet checks if the configuration has reached step 2 and, if so, runs the **motd** Puppet class.

6.3. INCLUDING A USER-DEFINED COMPOSABLE SERVICES

You can configure the custom motd service only on the overcloud Controller nodes. This requires a custom environment file and custom roles data file included with your deployment. Replace example input in this procedure according to your requirements.

Procedure

1. Add the new service to an environment file, **env-motd.yaml**, as a registered heat resource within the **OS::TripleO::Services** namespace. In this example, the resource name for our **motd** service is **OS::TripleO::Services::Motd**:

```
resource_registry:
  OS::TripleO::Services::Motd: /home/stack/templates/motd.yaml

parameter_defaults:
  MotdMessage: |
    You have successfully accessed my Red Hat OpenStack Platform environment!
```

   This custom environment file also includes a new message that overrides the default for **MotdMessage**.

   The deployment now includes the **motd** service. However, each role that requires this new service must have an updated **ServicesDefault** listing in a custom **roles_data.yaml** file.

2. Create a copy of the default **roles_data.yaml** file:

```
$ cp /usr/share/openstack-tripleo-heat-templates/roles_data.yaml ~/custom_roles_data.yaml
```

3. To edit this file, scroll to the **Controller** role and include the service in the **ServicesDefault** listing:

```
- name: Controller
  CountDefault: 1
```

```
  ServicesDefault:
    - OS::TripleO::Services::CACerts
    - OS::TripleO::Services::CephMon
    - OS::TripleO::Services::CephExternal
...
    - OS::TripleO::Services::FluentdClient
    - OS::TripleO::Services::VipHosts
    - OS::TripleO::Services::Motd           # Add the service to the end
```

4. When you create an overcloud, include the resulting environment file and the **custom_roles_data.yaml** file with your other environment files and deployment options:

```
$ openstack overcloud deploy --templates -e /home/stack/templates/env-motd.yaml -r
~/custom_roles_data.yaml [OTHER OPTIONS]
```

CHAPTER 7. BUILDING CERTIFIED CONTAINER IMAGES

You can use the partner **Build Service** to build your application containers for certification. The **Build Service** builds containers from Git repositories that are Internet-accessible publicly or privately with an SSH key.

Use the automated **Build Service** as part of **Red Hat OpenStack and NFV Zone** to automatically build containerized partner platform plugins to Red Hat OpenStack Platform (RHOSP) 16.2 base containers.

Prerequisites

- Register with Red Hat Connect for Technology Partners.
- Apply for Zone access to the Red Hat OpenStack and NFV zone.
- Create a Product. The information you provide is used when the certification is published in the Red Hat catalog.
- Create a git repository for your plugin, with your Dockerfile and any components that you want to include in the container.

If you have any problems when you register with or access the Red Hat Connect site, contact the Red Hat Technology Partner Success Desk.

7.1. ADDING A CONTAINER PROJECT

One project represents one partner image. If you have multiple images, you must create multiple projects.

Procedure

1. Log in to Red Hat Connect for Technology Partners and click **Zones**.
2. Scroll down and select the **Red Hat OpenStack & NFV** zone. Click anywhere in the box.
3. Click **Certify** to access the existing products and projects of your company.
4. Click **Add Project** to create a new project.
5. Set the **Project Name**.
    - The project name is not visible outside the system.
    - The project name must include the following elements: **[product][version]-[extended-base-container-image]-[your-plugin]**
    - For Red Hat OpenStack Platform (RHOSP) purposes, the format is **rhospXX-baseimage-myplugin**.
    - Example: **rhosp16-openstack-cinder-volume-myplugin**
6. Select the **Product**, **Product Version**, and **Release Category** based on your product or plugin, and its version.
    - Create the product and its version prior to creating projects.

- Set the label release category to **Tech Preview**. Generally Available is not an option until you have completed API testing with Red Hat Certification. Refer to the plugin certification requirements when you have certified your container image.

7. Select the **Red Hat Product** and **Red Hat Product Version** based on the base image that you want to modify with your partner plugin. For this release, select **Red Hat OpenStack Platform** and **16.2**.

8. Click **Submit** to create the new project.

Result:

Red Hat assesses and confirms the certification of your project.

Send an email to connect@redhat.com stating whether the plugin is **in tree** or **out of tree** in regards to the upstream code.

- **In Tree** means the plugin is included in the OpenStack upstream code base and the plugin image is built by Red Hat and distributed with Red Hat OpenStack Platform 16.2.

- **Out of Tree** means the plugin image is not included in the OpenStack upstream code base and not distributed within RHOSP 16.2.

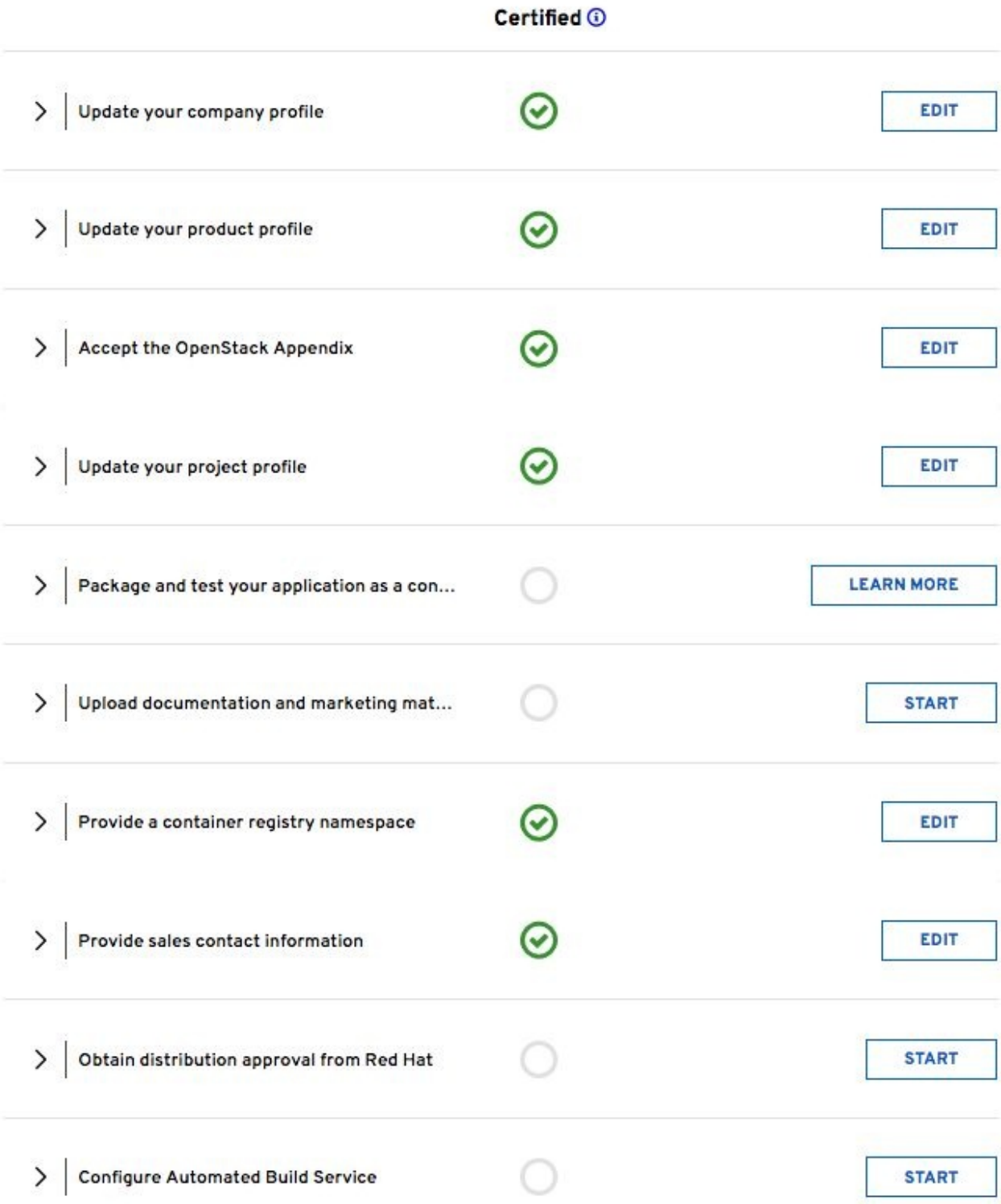
7.2. FOLLOWING THE CONTAINER CERTIFICATION CHECKLIST

Certified containers meet Red Hat standards for packaging, distribution, and maintenance. Containers that are certified by Red Hat have a high level of trust and supportability from container-capable platforms, including Red Hat OpenStack Platform (RHOSP). To maintain this, partners must keep their images up-to-date.

Procedure

1. Click **Certification Checklist**.

2. Complete all sections of the checklist. If you need more information about an item on the checklist, click the drop-down arrow on the left to view the item information and links to other resources.



The checklist includes the following items:

Update your company profile

Ensures that your company profile is up to date.

Update your product profile

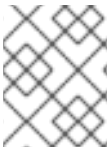
This page details to the product profile, including the product type, description, repository URL, version, and contact distribution list.

Accept the OpenStack Appendix

Site Agreement for the Container Terms.

Update project profile

Check that the image settings such as auto publish, registry namespace, release category, supported platforms are correct.



NOTE

In the **Supported Platforms** section, you must select an option so that you can save other required fields on this page.

Package and test your application as a container

Follow the instructions on this page to configure the build service. The build service is dependent on the completion of the previous steps.

Upload documentation and marketing materials

This redirects you to the product page. Scroll to the bottom and click **Add new Collateral** to upload your product information.

NOTE

You must provide a minimum of three materials. The first material must be a **document** type.

Provide a container registry namespace

This is the same as the project page profile page.

Provide sales contact information

This information is the same as the company profile.

Obtain distribution approval from Red Hat

Red Hat provides approval for this step.

Configure Automated Build Service

The configuration information to perform the build and scan of the container image.

The last item in the checklist is **Configure Automated Build Service** Before you configure this service, you must ensure that your project contains a dockerfile that conforms to Red Hat certification standards.

7.3. DOCKERFILE REQUIREMENTS

As a part of the image build process, the build service scans your built image to ensure that it complies with Red Hat standards. Use the following guidelines as a basis for the dockerfile to include with your project:

- The base image must be a Red Hat image. Any images that use Ubuntu, Debian, and CentOS as a base do not pass the scanner.
- You must configure the required labels:
  - **name**
  - **maintainer**
  - **vendor**
  - **version**

- **release**
- **summary**

- You must include a software license as a text file within the image. Add the software license to the **licenses** directory at the root of your project.
- You must configure a user that is not the **root** user.

The following dockerfile example demonstrates the required information for the scan:

```
FROM registry.redhat.io/rhosp-rhel8/openstack-cinder-volume
MAINTAINER VenderX Systems Engineering <maintainer@vendorX.com>

###Required Labels
LABEL name="rhosp-rhel8/openstack-cinder-volume-vendorx-plugin" \
      maintainer="maintainer@vendorX.com" \
      vendor="VendorX" \
      version="3.7" \
      release="1" \
      summary="Red Hat OpenStack Platform 16.2 cinder-volume VendorX PluginY" \
      description="Red Hat OpenStack Platform 16.2 cinder-volume VendorX PluginY"


USER root

###Adding package
###repo exmple
COPY vendorX.repo /etc/yum.repos.d/vendorX.repo

###adding package with curl
RUN curl -L -o /verdorX-plugin.rpm http://vendorX.com/vendorX-plugin.rpm

###adding local package
COPY verdorX-plugin.rpm /

# Enable a repo to install a package
RUN dnf clean all
RUN yum-config-manager --enable openstack-16.2-for-rhel-8-x86_64-rpms
RUN dnf install -y vendorX-plugin
RUN yum-config-manager --disable openstack-16.2-for-rhel-8-x86_64-rpms

# Add required license as text file in Liceses directory (GPL, MIT, APACHE, Partner End User
Agreement, etc)
RUN mkdir /licenses
COPY licensing.txt /licenses

USER cinder
```

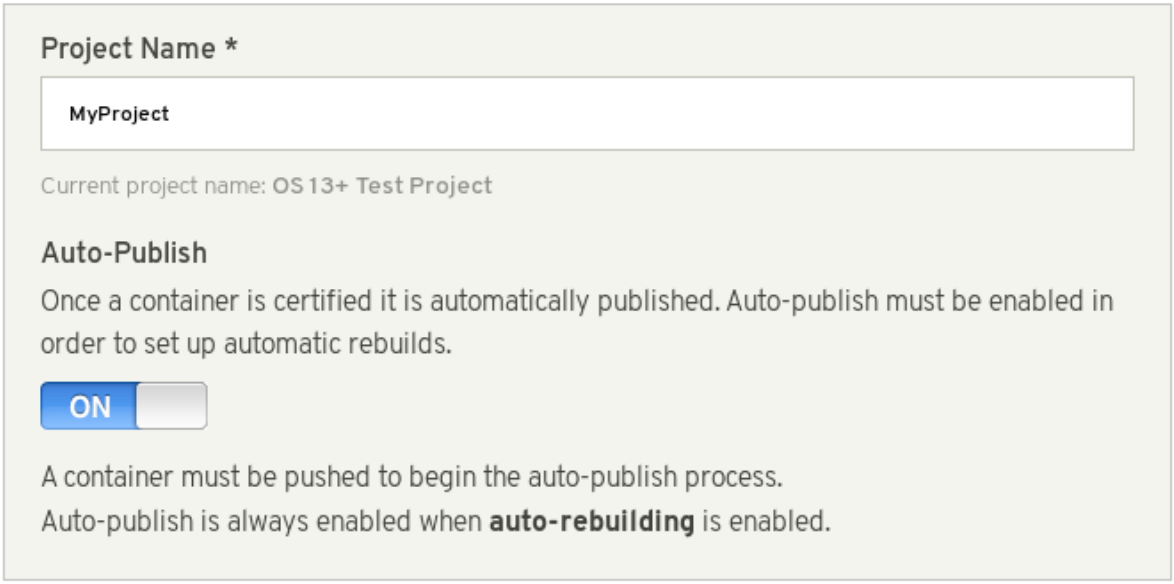
7.4. SETTING PROJECT DETAILS

You must set details for your project including the namespace and registry for your container image.

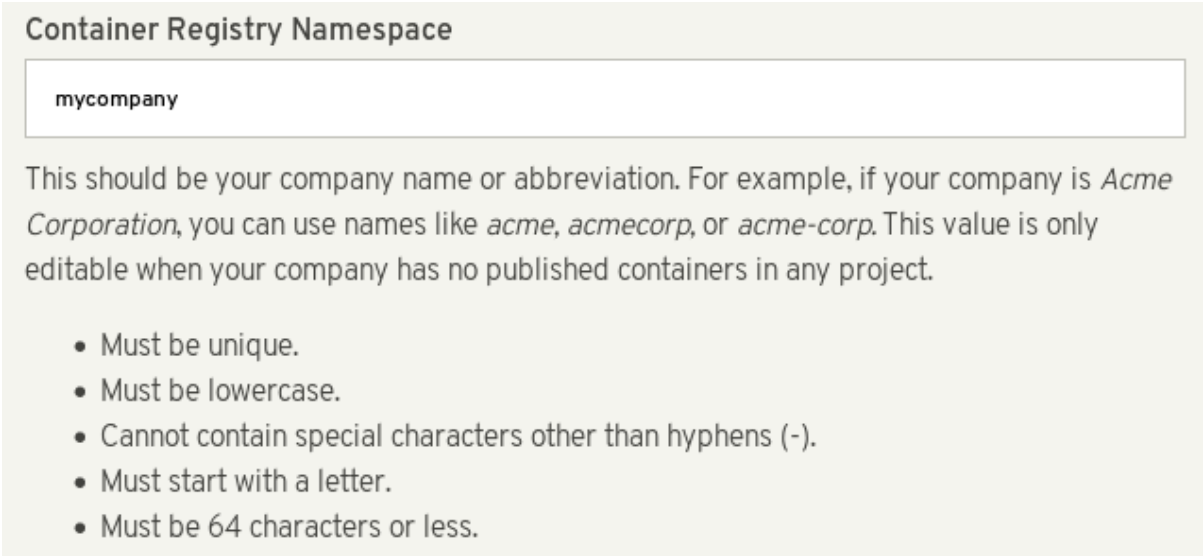
Procedure

1. Click **Project Settings**.

2. Ensure that your project name is in the correct format. Optionally, set **Auto-Publish** to **ON** if you want to automatically publish containers that pass certification. Certified containers are published in the Red Hat Container Catalog.

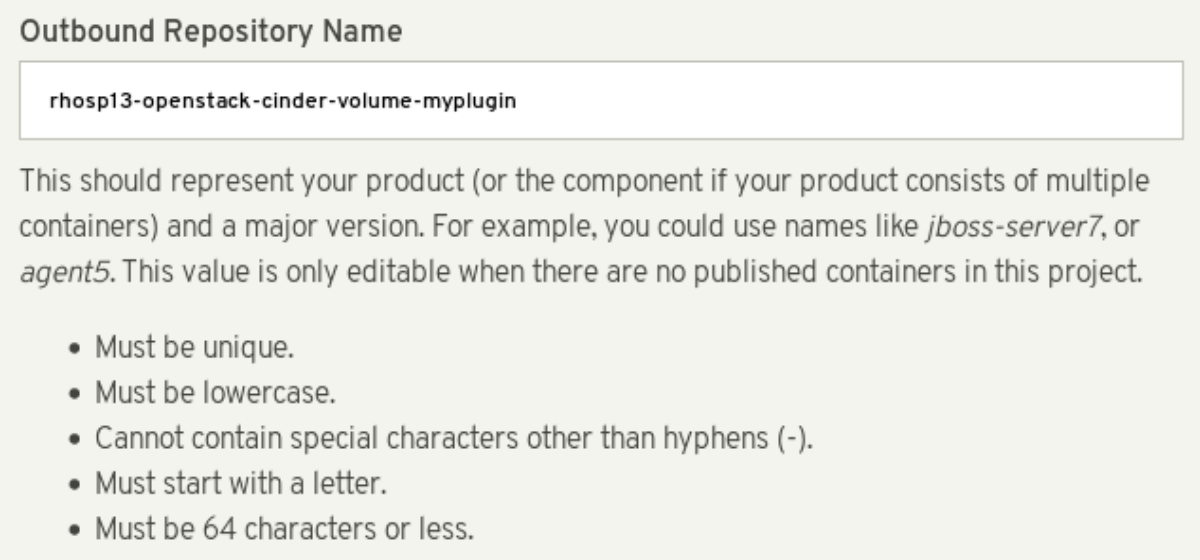


3. To set the **Container Registry Namespace**, follow the online instructions.



   - The container registry namespace is the name of your company.
   - The final registry URL is **registry.connect.redhat.com/namespace/repository:tag**.
   - Example: **registry.connect.redhat.com/mycompany/rhosp16-openstack-cinder-volume-myplugin:1.0**

4. To set the **Outbound Repository Name** and **Outbound Repository Descriptions**, follow the online instructions. The outbound repository name must be the same as the project name.



- **[product][version]-[extended_base_container_image]-[your_plugin]**
- For Red Hat OpenStack Platform (RHOSP) purposes, the format is **rhospXX-baseimage-myplugin**
- The final registry URL is **registry.connect.redhat.com/namespace/repository:tag**
- Example: **registry.connect.redhat.com/mycompany/rhosp16-openstack-cinder-volume-myplugin:1.0**

5. Add additional information about your project in the relevant fields:
   - **Repository Description**
   - **Supporting Documentation for Primed**
6. Click **Submit**.

7.5. BUILDING A CONTAINER IMAGE WITH THE BUILD SERVICE

Build the container image for your partner plugin.

Procedure

1. Click **Build Service**.
2. Click **Configure Build Service** to configure your build details.
   a. Ensure that the **Red Hat Container Build** is set to **ON**.
   b. Add your **Git Source URL** and optionally add your **Source Code SSH Key** if your git repository is protected. The URL can be HTML or SSH. SSH is required for protected git repositories.
   c. Optional: Add **Dockerfile Name** or leave blank if your Dockerfile name is **Dockerfile**.
   d. Optional: Add the **Context Directory** if the docker build context root is not the root of the git repository. Otherwise, leave this field blank.
   e. Set the **Branch** in your git repository to base the container image on.

f. Click **Submit** to finalize the **Build Service** settings.

3. Click **Start Build**.

4. Add a **Tag Name** and click **Submit**. It can take up to six minutes for the build to complete.

   - The tag name must be a version of your plugin.
   - The final reference URL is **registry.connect.redhat.com/namespace/repository:tag**.
   - Example: **registry.connect.redhat.com/mycompany/rhosp16-openstack-cinder-volume-myplugin:1.0**

5. Click **Refresh** to check that your build is complete. Optional: Click the matching **Build ID** to view the build details and logs.

6. The build service both builds and scans the image. This normally takes 10-15 minutes to complete. When the scan completes, click the **View** link to expand the scan results.

7.6. CORRECTING FAILED SCAN RESULTS

The **Scan Details** page displays the result of the scan, including any failed items. If your image scan reports a **FAILED** status, use the following procedure to investigate how to correct the failure.

Procedure

1. On the **Container Information** page, click the **View** link to expand the scan results.

2. Click the failed item. For example, in the following screenshot, the **has_licenses** check failed.

Scan Details

▾ Assessments

Name	Value
has_licenses	X
not_running_privileged	✓
rpm_list_successful	✓
rpm_verify_successful	✓
is_rhel	✓
vendor_label_exists	✓
free_of_critical_vulnerabilities	✓
good_tags	✓
good_layer_count	✓
release_label_exists	✓
not_running_as_root	✓
version_label_exists	✓
name_label_exists	✓

3. Click the failed item to open the **Policy Guide** at the relevant section and view more information about how to correct the issue.

NOTE

If you receive an **Access Denied** warning when you access the **Policy Guide**, email connect@redhat.com

7.7. PUBLISHING A CONTAINER IMAGE

After the container image passes the scan, you can publish the container image.

Procedure

1. On the **Container Information** page, click the **Publish** link to publish the container image live.
2. The **Publish** link changes to **Unpublish**. To unpublish a container, click the **Unpublish** link.

When you publish the link, check the certification documentation for more information about certifying your plugin. For more links to certification documentation, see Section 1.1, “Partner integration prerequisites”.

CHAPTER 8. INTEGRATION OF OPENSTACK COMPONENTS AND THEIR RELATIONSHIP WITH DIRECTOR AND THE OVERCLOUD

Use the following concepts about specific integration points to begin integrating hardware and software with Red Hat OpenStack Platform (RHOSP).

8.1. BARE METAL PROVISIONING (IRONIC)

Use the OpenStack Bare Metal Provisioning (ironic) component within director to control the power state of the nodes. Director uses a set of back-end drivers to interface with specific bare metal power controllers. These drivers are the key to enabling hardware and vendor specific extensions and capabilities. The most common driver is the IPMI driver, **pxe_ipmitool**, which controls the power state for any server that supports the Intelligent Platform Management Interface (IPMI).

Integration with Bare Metal Provisioning starts with the upstream OpenStack community. Ironic drivers accepted upstream are automatically included in the core RHOSP product and director by default. However, they might not be supported according to certification requirements.

Hardware drivers must undergo continuous integration testing to ensure their continued functionality. For more information about third-party driver testing and suitability, see the OpenStack community page Ironic Testing.

Upstream Repositories:

- OpenDev: https://opendev.org/openstack/ironic/

Puppet Module:

- GitHub: https://github.com/openstack/puppet-ironic

Bugzilla components:

- openstack-ironic
- python-ironicclient
- openstack-puppet-modules
- openstack-tripleo-heat-templates

Integration Notes:

- The upstream project contains drivers in the **ironic/drivers** directory.
- Director performs a bulk registration of nodes defined in a JSON file.
- Director is automatically configured to use Bare Metal Provisioning, which means the Puppet configuration requires little to no modification. However, if your driver is included with Bare Metal Provisioning, you must add your driver to the **/etc/ironic/ironic.yaml** file. Edit this file and search for the **IronicEnabledHardwareTypes** parameter:

  ```
  IronicEnabledHardwareTypes=ipmi,redfish,idrac,ilo
  ```

  This allows Bare Metal Provisioning to use the specified driver from the **drivers** directory.

8.2. NETWORKING (NEUTRON)

OpenStack Networking (neutron) provides the ability to create a network architecture within your cloud environment. The project provides several integration points for Software Defined Networking (SDN) vendors. These integration points usually fall into the categories of plugins or agents:

A plugin allows extension and customization of pre-existing neutron functions. Vendors can write plugins to ensure interoperability between neutron and certified software and hardware. Develop a driver for neutron Modular Layer 2 (ml2) plugin, which provides a modular back end for integrating your own drivers.

An agent provides a specific network function. The main neutron server and its plugins communicate with neutron agents. Existing examples include agents for DHCP, Layer 3 support, and bridging support.

For both plugins and agents, you can choose one of the following options:

- Include them for distribution as part of the Red Hat OpenStack Platform (RHOSP) solution
- Add them to the overcloud images after RHOSP distribution

Analyze the functionality of existing plugins and agents to determine how to integrate your own certified hardware and software. In particular, it is recommended to first develop a driver as a part of the ml2 plugin.

Upstream Repositories:

- OpenDev: https://opendev.org/openstack/neutron/

Upstream Blueprints:

- Launchpad: http://launchpad.net/neutron

Puppet Module:

- GitHub: https://github.com/openstack/puppet-neutron

Bugzilla components:

- openstack-neutron
- python-neutronclient
- openstack-puppet-modules
- openstack-tripleo-heat-templates

Integration Notes:

- The upstream **neutron** project contains several integration points:
  - The plugins are located in **neutron/plugins/**
  - The ml2 plugin drivers are located in **neutron/plugins/ml2/drivers/**
  - The agents are located in **neutron/agents/**

- Since the OpenStack Liberty release, many of the vendor-specific ml2 plugin have been moved into their own repositories beginning with **networking-**. For example, the Cisco-specific plugins are located in https://github.com/openstack/networking-cisco
- The **puppet-neutron** repository also contains separate directories to configure these integration points:
  - The plugin configuration is located in **manifests/plugins/**
  - The ml2 plugin driver configuration is located in **manifests/plugins/ml2/**
  - The agent configuration is located in **manifests/agents/**
- The **puppet-neutron** repository contains numerous additional libraries for configuration functions. For example, the **neutron_plugin_ml2** library adds a function to add attributes to the ml2 plugin configuration file.

8.3. BLOCK STORAGE (CINDER)

OpenStack Block Storage (cinder) provides an API that interacts with block storage devices, which Red Hat OpenStack Platform (RHOSP) uses to create volumes. For example, Block Storage provides virtual storage devices for instances. Block Storage provides a core set of drivers to support different storage hardware and protocols. For example, some of the core drivers include support for NFS, iSCSI, and Red Hat Ceph Storage. Vendors can include drivers to support additional certified hardware.

Vendors have the following two main options with the drivers and configuration they develop:

- Include them for distribution as part of the RHOSP solution.
- Add them to the overcloud images after RHOSP distribution.

Analyze the functionality of existing drivers to determine how to integrate your own certified hardware and software.

Upstream Repositories:

- OpenDev: https://opendev.org/openstack/cinder

Upstream Blueprints:

- Launchpad: http://launchpad.net/cinder

Puppet Module:

- GitHub: https://github.com/openstack/puppet-cinder

Bugzilla components:

- openstack-cinder
- python-cinderclient
- openstack-puppet-modules
- openstack-tripleo-heat-templates

Integration Notes:

- The upstream **cinder** repository contains the drivers in **cinder/volume/drivers/**
- The **puppet-cinder** repository contains two main directories for driver configuration:
  - The **manifests/backend** directory contains a set of defined types that configure the drivers.
  - The **manifests/volume** directory contains a set of classes to configure a default block storage device.
- The **puppet-cinder** repository contains a library called **cinder_config** to add attributes to the Cinder configuration files.

8.4. IMAGE STORAGE (GLANCE)

OpenStack Image service (glance) provides an API that interacts with storage types to provide storage for images. Image service provides a core set of drivers to support different storage hardware and protocols. For example, the core drivers include support for file, OpenStack Object Storage (swift), OpenStack Block Storage (cinder), and Red Hat Ceph Storage. Vendors can include drivers to support additional certified hardware.

Upstream Repositories:

- OpenDev:
  - https://opendev.org/openstack/glance/
  - https://opendev.org/openstack/glance_store/

Upstream Blueprints:

- Launchpad: http://launchpad.net/glance

Puppet Module:

- GitHub: https://github.com/openstack/puppet-glance

Bugzilla components:

- openstack-glance
- python-glanceclient
- openstack-puppet-modules
- openstack-tripleo-heat-templates

Integration Notes:

- Adding a vendor-specific driver is not necessary because Image service can use Block Storage, which contains integration points, to manage image storage.
- The upstream **glance_store** repository contains the drivers in **glance_store/_drivers**.
- The **puppet-glance** repository contains the driver configuration in the **manifests/backend** directory.

- The **puppet-glance** repository contains a library called **glance_api_config** to add attributes to the Glance configuration files.

8.5. SHARED FILE SYSTEMS (MANILA)

OpenStack Shared File Systems service (manila) provides an API for shared and distributed file system services. Vendors can include drivers to support additional certified hardware.

Upstream Repositories:

- OpenDev: https://opendev.org/openstack/manila/

Upstream Blueprints:

- Launchpad: http://launchpad.net/manila

Puppet Module:

- GitHub: https://github.com/openstack/puppet-manila

Bugzilla components:

- openstack-manila
- python-manilaclient
- openstack-puppet-modules
- openstack-tripleo-heat-templates

Integration Notes:s

- The upstream **manila** repository contains the drivers in **manila/share/drivers/**.
- The **puppet-manila** repository contains the driver configuration in the **manifests/backend** directory.
- The **puppet-manila** repository contains a library called **manila_config** to add attributes to the manila configuration files.

8.6. OPENSHIFT-ON-OPENSTACK

Red Hat OpenStack Platform (RHOSP) aims to support OpenShift-on-OpenStack deployments. For more information about the partner integration for these deployments, see the Red Hat OpenShift Partners page.

APPENDIX A. COMPOSABLE SERVICE PARAMETERS

The following parameters are used for the outputs in all composable services:

- service_name
- config_settings
- kolla_config
- docker_config
- puppet_config
- container_puppet_tasks
- global_config_settings
- service_config_settings
- step_config
- host_prep_tasks
- upgrade_tasks
- upgrade_batch_tasks
- post_upgrade_tasks
- update_tasks
- post_update_tasks
- external_deploy_tasks
- external_upgrade_tasks
- external_update_tasks

service_name

The name of your service. You can use this to apply configuration from other composable services with service_config_settings.

config_settings

Custom hieradata settings for your service.

kolla_config

Creates a map of the kolla configuration in the container. The format begins with the absolute path of the configuration file as a key and then uses the following sub-parameters as the value of the key:

command

The command to run when the container starts.

config_files

The location of the service configuration files (**source**) and the destination on the container (**dest**) before the service starts. Also includes options to either merge or replace these files on the container (**merge**), whether to preserve the file permissions and other properties (**preserve_properties**).

permissions

Set permissions for certain directories on the containers. Requires a **path** and an **owner** (which is a **user:group** combination). You can also apply the permissions recursively (**recurse**).

The following is an example of the **kolla_config** parameter for the keystone service:

```
kolla_config:
  /var/lib/kolla/config_files/keystone.json:
    command: /usr/sbin/httpd -DFOREGROUND
    config_files:
      - source: "/var/lib/kolla/config_files/src/*"
        dest: "/"
        merge: true
        preserve_properties: true
  /var/lib/kolla/config_files/keystone_cron.json:
    command: /usr/sbin/crond -n
    config_files:
      - source: "/var/lib/kolla/config_files/src/*"
        dest: "/"
        merge: true
        preserve_properties: true
    permissions:
      - path: /var/log/keystone
        owner: keystone:keystone
        recurse: true
```

docker_config

Data passed to the **paunch** command to configure containers at each step.

- **step_0** - Containers configuration files generated per hiera settings.
- **step_1** - Load Balancer configuration
    a. Baremetal configuration
    b. Container configuration
- **step_2** - Core Services (Database/Rabbit/NTP/etc.)
    a. Baremetal configuration
    b. Container configuration
- **step_3** - Early OpenStack Service setup (Ringbuilder, etc.)
    a. Baremetal configuration
    b. Container configuration
- **step_4** - General OpenStack Services
    a. Baremetal configuration

b. Container configuration

c. Keystone container post initialization (tenant, service, endpoint creation)

- **step_5** - Service activation (Pacemaker)

a. Baremetal configuration

b. Container configuration

The YAML file uses a set of parameters to define the container to run at each step and the **podman** settings associated with each container. For example:

```
docker_config:
  step_3:
    keystone:
      start_order: 2
      image: *keystone_image
      net: host
      privileged: false
      restart: always
      healthcheck:
        test: /openstack/healthcheck
      volumes: *keystone_volumes
      environment:
        - KOLLA_CONFIG_STRATEGY=COPY_ALWAYS
```

This creates a **keystone** container and uses the respective parameters to define details, including the image to use, the networking type, and environment variables.

puppet_config

This section is a nested set of key value pairs that drive the creation of configuration files using Puppet. The following required parameters are included:

puppet_tags

Puppet resource tag names that are used to generate configuration files with Puppet. Only the named configuration resources are used to generate a file. Any service that specifies tags has the default tags of *file,concat,file_line,augeas,cron* appended to the setting. Example: **keystone_config**

config_volume

The name of the volume (directory) where the configuration files are generated for this service. Use this as the location to bind mount into the running Kolla container for configuration.

config_image

The name of the container image that will be used for generating configuration files. This is often the same container that the runtime service uses. Some services share a common set of configuration files which are generated in a common base container.

step_config

This setting controls the manifest that is used to create configuration files with Puppet. Use the following Puppet tags together with the manifest to generate a configuration directory for this container.

container_puppet_tasks

Provides data to drive the **container-puppet.py** tool directly. The task is executed once within the cluster, not on each node, and is useful for several Puppet snippets required for initialization of things like keystone endpoints and database users. For example:

```
container_puppet_tasks:
  # Keystone endpoint creation occurs only on single node
  step_3:
    config_volume: 'keystone_init_tasks'
    puppet_tags:
'keystone_config,keystone_domain_config,keystone_endpoint,keystone_identity_provider,keystone_pas
e_ini,keystone_role,keystone_service,keystone_tenant,keystone_user,keystone_user_role,keystone_do
main'
    step_config: 'include ::tripleo::profile::base::keystone'
    config_image: *keystone_config_image
```

global_config_settings

Custom hieradata settings distributed to all roles.

service_config_settings

Custom hieradata settings for another service. For example, your service might require that its endpoints are registered in OpenStack Identity (keystone). This provides parameters from one service to another and provides a method of cross-service configuration, even if the services are on different roles.

step_config

This is a Puppet snippet to configure the service. This snippet is added to a combined manifest created and run at each step of the service configuration process:

- Step 1 - Load balancer configuration
- Step 2 - Core high availability and general services (Database, RabbitMQ, NTP)
- Step 3 - Early OpenStack Platform Service setup (Storage, Ring Building)
- Step 4 - General OpenStack Platform services
- Step 5 - Service activation (Pacemaker) and OpenStack Identity (keystone) role and user creation

In any referenced puppet manifest, you can use the **step** hieradata (using **hiera('step')**) to define specific actions at specific steps during the deployment process.

host_prep_tasks

This is an ansible snippet to execute on the node host to prepare it for containerized services. For example, you might need to create a specific directory to mount to the container during its creation.

external_deploy_tasks

Execute Ansible tasks on the undercloud and run at each **step** in the deployment process.

upgrade_tasks

This is an ansible snippet to help with upgrading the service. The snippet is added to a combined playbook. Each operation uses a tag to define a **step**, which includes the following items:

- **common** - Applies to all steps
- **step0** - Validation
- **step1** - Stop all OpenStack services.
- **step2** - Stop all Pacemaker-controlled services
- **step3** - Package update and new package installation
- **step4** - Start OpenStack service required for database upgrade
- **step5** - Upgrade database

upgrade_batch_tasks

Performs a similar function to **upgrade_tasks** but only executes a batch set of Ansible tasks in the order they are listed. The default is **1**, but you can change this per role by using the **upgrade_batch_size** parameter in a **roles_data.yaml** file.

post_upgrade_tasks

Executes Ansible tasks after the completion of the upgrade process.

external_upgrade_tasks

Execute Ansible tasks on the undercloud and run at each **step** in the upgrade process.

update_tasks

Ansible snippet to help with minor version updates for a service. The snippet is added to a combined playbook. Each operation uses a tag to define a **step**, which includes the following items:

- **common** - Applies to all steps
- **step0** - Validation
- **step1** - Stop all OpenStack services.
- **step2** - Stop all Pacemaker-controlled services
- **step3** - Package update and new package installation
- **step4** - Start OpenStack service required for database upgrade
- **step5** - Upgrade database

post_update_tasks

Executes Ansible tasks after the completion of the update process.

external_update_tasks

Execute Ansible tasks on the undercloud and run at each **step** in the minor version update process.