Red Hat OpenStack Platform 14

Transitioning to Containerized Services

A basic guide to working with OpenStack Platform containerized services

Last Updated: 2019-02-11

Red Hat OpenStack Platform 14 Transitioning to Containerized Services

A basic guide to working with OpenStack Platform containerized services

OpenStack Team
rhos-docs@redhat.com

Abstract

This guide provides some basic information to help users get accustomed working with OpenStack Platform services running in containers.

Table of Contents

CHAPTER 1. INTRODUCTION

Past versions of Red Hat OpenStack Platform used services managed with Systemd. However, more recent version of OpenStack Platform now use containers to run services. Some administrators might not have a good understanding of how containerized OpenStack Platform services operate, and so this guide aims to help you understand OpenStack Platform container images and containerized services. This includes:

- How to obtain and modify container images
- How to manage containerized services in the overcloud
- Understanding how containers differ from Systemd services

The main goal is to help you gain enough knowledge of containerized OpenStack Platform services to transition from a Systemd-based environment to a container-based environment.

1.1. CONTAINERIZED SERVICES AND KOLLA

Each of the main Red Hat OpenStack Platform services run in containers. This provides a method of keep each service within its own isolated namespace separated from the host. This means:

- The deployment of services is performed by pulling container images from the Red Hat Custom Portal and running them.
- The management functions, like starting and stopping services, operate through the **`docker`** command.
- Upgrading containers require pulling new container images and replacing the existing containers with newer versions.

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

CHAPTER 2. OBTAINING AND MODIFYING CONTAINER IMAGES

A containerized overcloud requires access to a registry with the required container images. This chapter provides information on how to prepare the registry and your undercloud and overcloud configuration to use container images for Red Hat OpenStack Platform.

2.1. PREPARING CONTAINER IMAGES

The overcloud configuration requires some 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 any modifications to suit your needs.

2.2. 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 lists 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.
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.
pull_source	The source registry to pull the original container images from.
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 lists the keys and their descriptions:

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.
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 the director uses to identify the images to pull from the source registry. You usually keep this key set to **latest**.

NOTE

The **set** section might contains several parameters that begin with **openshift_**. These parameters are for various scenarios involving OpenShift-on-OpenStack.

2.3. LAYERING IMAGE PREPARATION ENTRIES

The value of ContainerImagePrepare 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 **14.0-44**:

```
ContainerImagePrepare:
- tag_from_label: "{version}-{release}"
  push_destination: true
  excludes:
  - nova-api
  set:
    namespace: registry.access.redhat.com/rhosp14
    name_prefix: openstack-
    name_suffix: ''
    tag: latest
- push_destination: true
  includes:
    - nova-api
    set:
      namespace: registry.access.redhat.com/rhosp14
      tag: 14.0-44
```

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.

2.4. MODIFYING IMAGES DURING PREPARATION

It is possible to modify images during image preparation to make any required changes, 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 addons 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, then tags the result. The prepare command can then 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 via 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 being modified as expected:

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

2.5. 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
  ...
```

2.6. INSTALLING ADDITIONAL RPM FILES TO CONTAINER IMAGES

You can also 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
  ...
```

2.7. 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/rhosp14/openstack-nova-compute:latest

USER "root"

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

USER "nova"
```

2.8. PREPARING A SATELLITE SERVER FOR CONTAINER IMAGES

Red Hat Satellite 6 offers registry synchronization capabilities. This provides a method to pull multiple images into a Satellite server and manage them as part of an application life cycle. The Satellite also acts as a registry for other container-enabled systems to use. For more details information on managing container images, see "Managing Container Images" in the *Red Hat Satellite 6 Content Management Guide*.

The examples in this procedure use the **hammer** command line tool for Red Hat Satellite 6 and an example organization called **ACME**. Substitute this organization for your own Satellite 6 organization.

Procedure

1. Create a list of all container images, including the Ceph images:

```
$ sudo docker search "registry.access.redhat.com/rhosp14" | awk '{
print $2 }' | grep -v beta | sed "s/registry.access.redhat.com\///g"
| tail -n+2 > satellite_images
$ echo "rhceph/rhceph-3-rhel7" >> satellite_images_names
```

2. Copy the **satellite_images_names** file to a system that contains the Satellite 6 **hammer** tool. Alternatively, use the instructions in the *Hammer CLI Guide* to install the **hammer** tool to the undercloud.

3. Run the following **hammer** command to create a new product (**OSP14 Containers**) in your Satellite organization:

```
$ hammer product create \
  --organization "ACME" \
  --name "OSP14 Containers"
```

This custom product will contain our images.

4. Add the base container image to the product:

```
$ hammer repository create \
  --organization "ACME" \
  --product "OSP14 Containers" \
  --content-type docker \
  --url https://registry.access.redhat.com \
  --docker-upstream-name rhosp14/openstack-base \
  --name base
```

_

5. Add the overcloud container images from the **`satellite_images`** file.

```
$ while read IMAGE; do \
  IMAGENAME=$(echo $IMAGE | cut -d"/" -f2 | sed "s/openstack-//g" |
sed "s/:.*//g") ; \
  hammer repository create \
  --organization "ACME" \
  --product "OSP14 Containers" \
  --content-type docker \
  --url https://registry.access.redhat.com \
  --docker-upstream-name $IMAGE \
  --name $IMAGENAME ; done < satellite_images_names
```

6. Synchronize the container images:

```
$ hammer product synchronize \
  --organization "ACME" \
  --name "OSP14 Containers"
```

   Wait for the Satellite server to complete synchronization.

   NOTE

   Depending on your configuration, **`hammer`** might ask for your Satellite server username and password. You can configure **`hammer`** to automatically login using a configuration file. For more information, see the "Authentication" section in the *Hammer CLI Guide*.

7. If your Satellite 6 server uses content views, create a new content view version to incorporate the images and promote it along environments in your application life cycle. This largely depends on how you structure your application lifecycle. For example, if you have an environment called **`production`** in your lifecycle and you want the container images available in that environment, create a content view that includes the container images and promote that content view to the **`production`** environment. For more information, see "Managing Container Images with Content Views".

8. Check the available tags for the **`base`** image:

```
$ hammer docker tag list --repository "base" \
  --organization "ACME" \
  --environment "production" \
  --content-view "myosp14" \
  --product "OSP14 Containers"
```

   This command displays tags for the OpenStack Platform container images within a content view for an particular environment.

9. Return to the undercloud and generate a default environment file for preparing images using your Satellite server as a source. Run the following example command to generate the environment file:

```
(undercloud) $ openstack tripleo container image prepare default \
  --output-env-file containers-prepare-parameter.yaml
```

- **`--output-env-file`** is an environment file name. The contents of this file will include the parameters for preparing your container images for the undercloud. In this case, the name of the file is **`containers-prepare-parameter.yaml`**.

10. Edit the **`containers-prepare-parameter.yaml`** file and modify the following parameters:

    - **`namespace`** - The URL and port of the registry on the Satellite server. The default registry port on Red Hat Satellite is 5000.
    - **`name_prefix`** - The prefix is based on a Satellite 6 convention. This differs depending on whether you use content views:
        - If you use content views, the structure is **`[org]-[environment]-[content view]-[product]-`**. For example: **`acme-production-myosp14-osp14_containers-`**.
        - If you do not use content views, the structure is **`[org]-[product]-`**. For example: **`acme-osp14_containers-`**.
    - **`ceph_namespace`**, **`ceph_image`**, **`ceph_tag`** - If using Ceph Storage, include the additional parameters to define the Ceph Storage container image location. Note that **`ceph_image`** now includes a Satellite-specific prefix. This prefix is the same value as the **`name_prefix`** option.

The following example environment file contains Satellite-specific parameters:

```
parameter_defaults:
  ContainerImagePrepare:
  - push_destination: true
    set:
      ceph_image: acme-production-myosp14-osp14_containers-rhceph-3-rhel7
      ceph_namespace: satellite.example.com:5000
      ceph_tag: latest
      name_prefix: acme-production-myosp14-osp14_containers-
      name_suffix: ''
      namespace: satellite.example.com:5000
      neutron_driver: null
      tag: latest
      ...
    tag_from_label: '{version}-{release}'
```

Use this environment file when creating both your undercloud and overcloud.

CHAPTER 3. INSTALLING THE UNDERCLOUD WITH CONTAINERS

This chapter provides info on how to create a container-based undercloud and keep it updated.

3.1. CONFIGURING THE DIRECTOR

The director installation process requires certain settings in the **`undercloud.conf`** configuration file, which the director reads from the **`stack`** user's home directory. This procedure demonstrates how to use the default template as a foundation for your configuration.

Procedure

1. Copy the default template to the **`stack`** user's home directory:

```
[stack@director ~]$ cp \
  /usr/share/python-tripleoclient/undercloud.conf.sample \
  ~/undercloud.conf
```

2. Edit the **`undercloud.conf`** file. This file contains settings to configure your undercloud. If you omit or comment out a parameter, the undercloud installation uses the default value.

3.2. DIRECTOR CONFIGURATION PARAMETERS

The following list contains information about parameters for configuring the **`undercloud.conf`** file. Keep all parameters within their relevant sections to avoid errors.

Defaults

The following parameters are defined in the **`[DEFAULT]`** section of the **`undercloud.conf`** file:

additional_architectures

A list of additional (kernel) architectures that an overcloud supports. Currently the overcloud supports **`ppc64le`** architecture.

> **NOTE**
>
> When enabling support for ppc64le, you must also set **`ipxe_enabled`** to **`False`**

certificate_generation_ca

The **`certmonger`** nickname of the CA that signs the requested certificate. Use this option only if you have set the **`generate_service_certificate`** parameter. If you select the **`local`** CA, certmonger extracts the local CA certificate to **`/etc/pki/ca-trust/source/anchors/cm-local-ca.pem`** and adds the certificate to the trust chain.

clean_nodes

Defines whether to wipe the hard drive between deployments and after introspection.

cleanup

Cleanup temporary files. Set this to **`False`** to leave the temporary files used during deployment in place after the command is run. This is useful for debugging the generated files or if errors occur.

container_images_file

Heat environment file with container image information. This can either be:

- Parameters for all required container images
- Or the **`ContainerImagePrepare`** parameter to drive the required image preparation. Usually the file containing this parameter is named **`containers-prepare-parameter.yaml`**.

custom_env_files

Additional environment file to add to the undercloud installation.

deployment_user

The user installing the undercloud. Leave this parameter unset to use the current default user (**`stack`**).

discovery_default_driver

Sets the default driver for automatically enrolled nodes. Requires **`enable_node_discovery`** enabled and you must include the driver in the **`enabled_hardware_types`** list.

docker_insecure_registries

A list of insecure registries for **`docker`** to use. Use this parameter if you want to pull images from another source, such as a private container registry. In most cases, docker has the certificates to pull container images from either the Red Hat Container Catalog or from your Satellite server if the undercloud is registered to Satellite.

docker_registry_mirror

An optional **`registry-mirror`** configured in **`/etc/docker/daemon.json`**

enable_ironic; enable_ironic_inspector; enable_mistral; enable_tempest; enable_validations; enable_zaqar

Defines the core services to enable for director. Leave these parameters set to **`true`**.

enable_ui

Defines whether to install the director web UI. Use this parameter to perform overcloud planning and deployments through a graphical web interface. Note that the UI is only available with SSL/TLS enabled using either the **`undercloud_service_certificate`** or **`generate_service_certificate`**.

enable_node_discovery

Automatically enroll any unknown node that PXE-boots the introspection ramdisk. New nodes use the **`fake_pxe`** driver as a default but you can set **`discovery_default_driver`** to override. You can also use introspection rules to specify driver information for newly enrolled nodes.

enable_novajoin

Defines whether to install the **`novajoin`** metadata service in the Undercloud.

enable_routed_networks

Defines whether to enable support for routed control plane networks.

enable_swift_encryption

Defines whether to enable Swift encryption at-rest.

enable_telemetry

Defines whether to install OpenStack Telemetry services (gnocchi, aodh, panko) in the undercloud. Set **`enable_telemetry`** parameter to **`true`** if you want to install and configure telemetry services automatically. The default value is **`false`**, which disables telemetry on the undercloud. This parameter is required if using other products that consume metrics data, such as Red Hat CloudForms.

enabled_hardware_types

A list of hardware types to enable for the undercloud.

generate_service_certificate

Defines whether to generate an SSL/TLS certificate during the undercloud installation, which is used for the **`undercloud_service_certificate`** parameter. The undercloud installation saves the resulting certificate **`/etc/pki/tls/certs/undercloud-[undercloud_public_vip].pem`**. The CA defined in the **`certificate_generation_ca`** parameter signs this certificate.

heat_container_image

URL for the heat container image to use. Leave unset.

heat_native

Use native heat templates. Leave as **`true`**.

hieradata_override

Path to **`hieradata`** override file. If set, the undercloud installation copies this file to the **`/etc/puppet/hieradata`** directory and sets it as the first file in the hierarchy. Use this parameter to provide custom configuration to services beyond the **`undercloud.conf`** parameters.

inspection_extras

Defines whether to enable extra hardware collection during the inspection process. This parameter requires **`python-hardware`** or **`python-hardware-detect`** package on the introspection image.

inspection_interface

The bridge the director uses for node introspection. This is a custom bridge that the director configuration creates. The **`LOCAL_INTERFACE`** attaches to this bridge. Leave this as the default **`br-ctlplane`**.

inspection_runbench

Runs a set of benchmarks during node introspection. Set this parameter to **`true`** to enable the benchmarks. This option is necessary if you intend to perform benchmark analysis when inspecting the hardware of registered nodes.

ipa_otp

Defines the one time password to register the Undercloud node to an IPA server. This is required when **`enable_novajoin`** is enabled.

ipxe_enabled

Defines whether to use iPXE or standard PXE. The default is **`true`**, which enables iPXE. Set to **`false`** to set to standard PXE.

local_interface

The chosen interface for the director’s Provisioning NIC. This is also the device the director uses for DHCP and PXE boot services. Change this value to your chosen device. To see which device is connected, use the **`ip addr`** command. For example, this is the result of an **`ip addr`** command:

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP qlen 1000
    link/ether 52:54:00:75:24:09 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.178/24 brd 192.168.122.255 scope global dynamic
eth0
       valid_lft 3462sec preferred_lft 3462sec
    inet6 fe80::5054:ff:fe75:2409/64 scope link
       valid_lft forever preferred_lft forever
```

```
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noop state
DOWN
    link/ether 42:0b:c2:a5:c1:26 brd ff:ff:ff:ff:ff:ff
```

In this example, the External NIC uses **`eth0`** and the Provisioning NIC uses **`eth1`**, which is currently not configured. In this case, set the **`local_interface`** to **`eth1`**. The configuration script attaches this interface to a custom bridge defined with the **`inspection_interface`** parameter.

local_ip

The IP address defined for the director's Provisioning NIC. This is also the IP address that the director uses for DHCP and PXE boot services. Leave this value as the default **`192.168.24.1/24`** unless you use a different subnet for the Provisioning network, for example, if it conflicts with an existing IP address or subnet in your environment.

local_mtu

MTU to use for the **`local_interface`**.

local_subnet

The local subnet to use for PXE boot and DHCP interfaces. The **`local_ip`** address should reside in this subnet. The default is **`ctlplane-subnet`**.

net_config_override

Path to network configuration override template. If you set this parameter, the undercloud uses a JSON format template to configure the networking with **`os-net-config`**. The undercloud ignores the network parameters set in **`undercloud.conf`**. See **`/usr/share/python-tripleoclient/undercloud.conf.sample`** for an example.

output_dir

Directory to output state, processed heat templates, and Ansible deployment files.

overcloud_domain_name

The DNS domain name to use when deploying the overcloud.

NOTE

When configuring the overcloud, the **`CloudDomain`** parameter must be set to a matching value. Set this parameter in an environment file when you configure your overcloud.

roles_file

The roles file to override for undercloud installation. It is highly recommended to leave unset so that the director installation uses the default roles file.

scheduler_max_attempts

Maximum number of times the scheduler attempts to deploy an instance. This value must be greater or equal to the number of bare metal nodes that you expect to deploy at once to work around potential race condition when scheduling.

service_principal

The Kerberos principal for the service using the certificate. Use this parameter only if your CA requires a Kerberos principal, such as in FreeIPA.

subnets

List of routed network subnets for provisioning and introspection. See Subnets for more information. The default value includes only the **`ctlplane-subnet`** subnet.

templates

Heat templates file to override.

undercloud_admin_host

The IP address defined for the director Admin API when using SSL/TLS. This is an IP address for administration endpoint access over SSL/TLS. The director configuration attaches the director's IP address to its software bridge as a routed IP address, which uses the **`/32`** netmask.

undercloud_debug

Sets the log level of undercloud services to **`DEBUG`**. Set this value to **`true`** to enable.

undercloud_enable_selinux

Enable or disable SELinux during the deployment. It is highly recommended to leave this value set to **`true`** unless you are debugging an issue.

undercloud_hostname

Defines the fully qualified host name for the undercloud. If set, the undercloud installation configures all system host name settings. If left unset, the undercloud uses the current host name, but the user must configure all system host name settings appropriately.

undercloud_log_file

The path to a log file to store the undercloud install/upgrade logs. By default, the log file is **`install-undercloud.log`** within the home directory. For example, **`/home/stack/install-undercloud.log`**.

undercloud_nameservers

A list of DNS nameservers to use for the undercloud hostname resolution.

undercloud_ntp_servers

A list of network time protocol servers to help synchronize the undercloud date and time.

undercloud_public_host

The IP address defined for the director Public API when using SSL/TLS. This is an IP address for accessing the director endpoints externally over SSL/TLS. The director configuration attaches this IP address to the director software bridge as a routed IP address, which uses the **`/32`** netmask.

undercloud_service_certificate

The location and filename of the certificate for OpenStack SSL/TLS communication. Ideally, you obtain this certificate from a trusted certificate authority. Otherwise, generate your own self-signed certificate.

undercloud_update_packages

Defines whether to update packages during the undercloud installation.

Subnets

Each provisioning subnet is a named section in the **`undercloud.conf`** file. For example, to create a subnet called **`ctlplane-subnet`**, use the following sample in your **`undercloud.conf`** file:

```
[ctlplane-subnet]
cidr = 192.168.24.0/24
dhcp_start = 192.168.24.5
dhcp_end = 192.168.24.24
inspection_iprange = 192.168.24.100,192.168.24.120
gateway = 192.168.24.1
masquerade = true
```

You can specify as many provisioning networks as necessary to suit your environment.

gateway

The gateway for the overcloud instances. This is the undercloud host, which forwards traffic to the External network. Leave this as the default **`192.168.24.1`** unless you use a different IP address for the director or want to use an external gateway directly.

> **NOTE**
>
> The director configuration also enables IP forwarding automatically using the relevant **`sysctl`** kernel parameter.

cidr

The network that the director uses to manage overcloud instances. This is the Provisioning network, which the undercloud **`neutron`** service manages. Leave this as the default **`192.168.24.0/24`** unless you use a different subnet for the Provisioning network.

masquerade

Defines whether to masquerade the network defined in the **`cidr`** for external access. This provides the Provisioning network with a degree of network address translation (NAT) so that the Provisioning network has external access through the director.

dhcp_start; dhcp_end

The start and end of the DHCP allocation range for overcloud nodes. Ensure this range contains enough IP addresses to allocate your nodes.

Modify the values for these parameters to suit your configuration. When complete, save the file.

3.3. INSTALLING THE DIRECTOR

Complete the following procedure to install the director and perform some basic post-installation tasks.

Procedure

1. Run the following command to install the director on the undercloud:

   ```
   [stack@director ~]$ openstack undercloud install
   ```

   This launches the director’s configuration script. The director installs additional packages and configures its services according to the configuration in the **`undercloud.conf`**. This script takes several minutes to complete.

   The script generates two files when complete:

   - **`undercloud-passwords.conf`** - A list of all passwords for the director’s services.
   - **`stackrc`** - A set of initialization variables to help you access the director’s command line tools.

2. The script also starts all OpenStack Platform service containers automatically. Check the enabled containers using the following command:

   ```
   [stack@director ~]$ sudo docker ps
   ```

3. The script adds the **`stack`** user to the **`docker`** group to give the **`stack`** user access to container management commands. Refresh the **`stack`** user’s permissions with the following command:

```
[stack@director ~]$ exec su -l stack
```

The command prompts you to log in again. Enter the stack user's password.

4. To initialize the **`stack`** user to use the command line tools, run the following command:

```
[stack@director ~]$ source ~/stackrc
```

The prompt now indicates OpenStack commands authenticate and execute against the undercloud;

```
(undercloud) [stack@director ~]$
```

The director installation is complete. You can now use the director's command line tools.

3.4. PERFORMING A MINOR UPDATE OF A CONTAINERIZED UNDERCLOUD

The director provides commands to update the packages on the undercloud node. This allows you to perform a minor update within the current version of your OpenStack Platform environment.

Procedure

1. Log into the director as the **`stack`** user.

2. Run **`yum`** to upgrade the director's main packages:

```
$ sudo yum update -y python-tripleoclient* openstack-tripleo-common
openstack-tripleo-heat-templates
```

3. The director uses the **`openstack undercloud upgrade`** command to update the undercloud environment. Run the command:

```
$ openstack undercloud upgrade
```

4. Wait until the undercloud upgrade process completes.

5. Reboot the undercloud to update the operating system's kernel and other system packages:

```
$ sudo reboot
```

6. Wait until the node boots.

CHAPTER 4. DEPLOYING AND UPDATING AN OVERCLOUD WITH CONTAINERS

This chapter provides info on how to create a container-based overcloud and keep it updated.

4.1. DEPLOYING AN OVERCLOUD

This procedure demonstrates how to deploy an overcloud with minimum configuration. The result will be a basic two-node overcloud (1 Controller node, 1 Compute node).

Procedure

1. Source the **`stackrc`** file:

```
$ source ~/stackrc
```

2. Run the **`deploy`** command and include the file containing your overcloud image locations (usually **`overcloud_images.yaml`**):

```
(undercloud) $ openstack overcloud deploy --templates \
  -e /home/stack/templates/overcloud_images.yaml \
  --ntp-server pool.ntp.org
```

3. Wait until the overcloud completes deployment.

4.2. UPDATING AN OVERCLOUD

For information on updating a containerized overcloud, see the Keeping Red Hat OpenStack Platform Updated guide.

CHAPTER 5. WORKING WITH CONTAINERIZED SERVICES

This chapter provides some examples of commands to manage containers and how to troubleshoot your OpenStack Platform containers

5.1. MANAGING CONTAINERIZED SERVICES

OpenStack Platform runs services in containers on the undercloud and overcloud nodes. In certain situations, you might need to control the individual services on a host. This section contains information about some common **`docker`** commands you can run on a node to manage containerized services. For more comprehensive information about using **`docker`** to manage containers, see "Working with Docker formatted containers" in the *Getting Started with Containers* guide.

Listing containers and images

To list running containers, run the following command:

```
$ sudo docker ps
```

To include stopped or failed containers in the command output, add the **`--all`** option to the command:

```
$ sudo docker ps --all
```

To list container images, run the following command:

```
$ sudo docker images
```

Inspecting container properties

To view the properties of a container or container images, use the **`docker inspect`** command. For example, to inspect the **`keystone`** container, run the following command:

```
$ sudo docker inspect keystone
```

Managing basic container operations

To restart a containerized service, use the **`docker restart`** command. For example, to restart the **`keystone`** container, run the following command:

```
$ sudo docker restart keystone
```

To stop a containerized service, use the **`docker stop`** command. For example, to stop the **`keystone`** container, run the following command:

```
$ sudo docker stop keystone
```

To start a stopped containerized service, use the **`docker start`** command. For example, to start the **`keystone`** container, run the following command:

```
$ sudo docker start keystone
```

NOTE

Any changes to the service configuration files within the container revert after restarting the container. This is because the container regenerates the service configuration based on files on the node's local file system in **`/var/lib/config-data/puppet-generated/`**. For example, if you edit **`/etc/keystone/keystone.conf`** within the **`keystone`** container and restart the container, the container regenerates the configuration using **`/var/lib/config-data/puppet-generated/keystone/etc/keystone/keystone.conf`** on the node's local file system, which overwrites any the changes made within the container before the restart.

Monitoring containers

To check the logs for a containerized service, use the **`docker logs`** command. For example, to view the logs for the **`keystone`** container, run the following command:

```
$ sudo docker logs keystone
```

Accessing containers

To enter the shell for a containerized service, use the **`docker exec`** command to launch **`/bin/bash`**. For example, to enter the shell for the **`keystone`** container, run the following command:

```
$ sudo docker exec -it keystone /bin/bash
```

To enter the shell for the **`keystone`** container as the root user, run the following command:

```
$ sudo docker exec --user 0 -it <NAME OR ID> /bin/bash
```

To exit from the container, run the following command:

```
# exit
```

Enabling swift-ring-builder on undercloud and overcloud

For continuity considerations in Object Storage (swift) builds, the **`swift-ring-builder`** and **`swift_object_server`** commands are no longer packaged on the undercloud or overcloud nodes. However, the commands are still available in the containers. To run them inside the respective containers:

```
docker exec -ti -u swift swift_object_server swift-ring-builder
/etc/swift/object.builder
```

If you require these commands, install the following package as the **`stack`** user on the undercloud or the **`heat-admin`** user on the overcloud:

```
sudo yum install -y python-swift
sudo yum install -y python2-swiftclient
```

5.2. TROUBLESHOOTING CONTAINERIZED SERVICES

If a containerized service fails during or after overcloud deployment, use the following recommendations to determine the root cause for the failure:

NOTE

Before running these commands, check that you are logged into an overcloud node and not running these commands on the undercloud.

Checking the container logs

Each container retains standard output from its main process. This output acts as a log to help determine what actually occurs during a container run. For example, to view the log for the **keystone** container, use the following command:

```
$ sudo docker logs keystone
```

In most cases, this log provides the cause of a container's failure.

Inspecting the container

In some situations, you might need to verify information about a container. For example, use the following command to view **keystone** container data:

```
$ sudo docker inspect keystone
```

This provides a JSON object containing low-level configuration data. You can pipe the output to the **jq** command to parse specific data. For example, to view the container mounts for the **keystone** container, run the following command:

```
$ sudo docker inspect keystone | jq .[0].Mounts
```

You can also use the **--format** option to parse data to a single line, which is useful for running commands against sets of container data. For example, to recreate the options used to run the **keystone** container, use the following **inspect** command with the **--format** option:

```
$ sudo docker inspect --format='{{range .Config.Env}} -e "{{.}}" {{end}}
{{range .Mounts}} -v {{.Source}}:{{.Destination}}{{if .Mode}}:{{.Mode}}
{{end}}{{end}} -ti {{.Config.Image}}' keystone
```

NOTE

The **--format** option uses Go syntax to create queries.

Use these options in conjunction with the **docker run** command to recreate the container for troubleshooting purposes:

```
$ OPTIONS=$( sudo docker inspect --format='{{range .Config.Env}} -e "
{{.}}" {{end}} {{range .Mounts}} -v {{.Source}}:{{.Destination}}{{if
.Mode}}:{{.Mode}}{{end}}{{end}} -ti {{.Config.Image}}' keystone )
$ sudo docker run --rm $OPTIONS /bin/bash
```

Running commands in the container

In some cases, you might need to obtain information from within a container through a specific Bash command. In this situation, use the following **`docker`** command to execute commands within a running container. For example, to run a command in the **`keystone`** container:

```
$ sudo docker exec -ti keystone <COMMAND>
```

NOTE

The **`-ti`** options run the command through an interactive pseudoterminal.

Replace **`<COMMAND>`** with your desired command. For example, each container has a health check script to verify the service connection. You can run the health check script for **`keystone`** with the following command:

```
$ sudo docker exec -ti keystone /openstack/healthcheck
```

To access the container's shell, run **`docker exec`** using **`/bin/bash`** as the command:

```
$ sudo docker exec -ti keystone /bin/bash
```

Exporting a container

When a container fails, you might need to investigate the full contents of the file. In this case, you can export the full file system of a container as a **`tar`** archive. For example, to export the **`keystone`** container's file system, run the following command:

```
$ sudo docker export keystone -o keystone.tar
```

This command create the **`keystone.tar`** archive, which you can extract and explore.

CHAPTER 6. COMPARING SYSTEMD SERVICES TO CONTAINERIZED SERVICES

This chapter provides some reference material to show how containerized services differ from Systemd services.

6.1. SYSTEMD SERVICE COMMANDS VS CONTAINERIZED SERVICE COMMANDS

The following table shows some similarities between Systemd-based commands and their Docker equivalents. This helps identify the type of service operation you aim to perform.

Function	Systemd-based	Docker-based
List all services	`systemctl list-units -t service`	`docker ps --all`
List active services	`systemctl list-units -t service --state active`	`docker ps`
Check status of service	`systemctl status openstack-nova-api`	`docker ps --filter "name=nova_api$" --all`
Stop service	`systemctl stop openstack-nova-api`	`docker stop nova_api`
Start service	`systemctl start openstack-nova-api`	`docker start nova_api`
Restart service	`systemctl restart openstack-nova-api`	`docker restart nova_api`

Show service configuration	`systemctl show openstack-nova-api`
	
	`systemctl cat openstack-nova-api`
		`docker inspect nova_api`

Show service logs	`journalctl -u openstack-nova-api`	`docker logs nova_api`

6.2. SYSTEMD SERVICES VS CONTAINERIZED SERVICES

The following table shows Systemd-based OpenStack services and their container-based equivalents.

OpenStack service	Systemd services	Docker containers

OpenStack service	Systemd services	Docker containers

aodh	`openstack-aodh-evaluator`
	`openstack-aodh-listener`
	`openstack-aodh-notifier`
	`httpd (openstack-aodh-api)`
		`aodh_listener`
		`aodh_api`
		`aodh_notifier`
		`aodh_evaluator`

ceilometer	`openstack-ceilometer-central`
	`openstack-ceilometer-collector`
	`openstack-ceilometer-notification`
	`httpd (openstack-ceilometer-api)`
		`ceilometer_agent_notification`
		`ceilometer_agent_central`

cinder	`openstack-cinder-api`
	`openstack-cinder-scheduler`
	`openstack-cinder-volume`
		`cinder_scheduler`
		`cinder_api`
		`openstack-cinder-volume-docker-0`

glance	`openstack-glance-api` `openstack-glance-registry`	`glance_api`

gnocchi	`openstack-gnocchi-metricd`
	`openstack-gnocchi-statsd`
	`httpd (openstack-gnocchi-api)`
		`gnocchi_statsd`
		`gnocchi_api`
		`gnocchi_metricd`

heat	`openstack-heat-api-cfn`
	`openstack-heat-api-cloudwatch`
	`openstack-heat-api`
	`openstack-heat-engine`
		`heat_api_cfn`
		`heat_engine`
		`heat_api`

OpenStack service	Systemd services	Docker containers
horizon	`httpd (openstack-dashboard)`	`horizon`
keystone	`httpd (openstack-keystone)`	`keystone`

neutron	`neutron-dhcp-agent`
	`neutron-l3-agent`
	`neutron-metadata-agent`
	`neutron-openvswitch-agent`
	`neutron-server`
		`neutron_ovs_agent`
		`neutron_l3_agent`
		`neutron_metadata_agent`
		`neutron_dhcp`
		`neutron_api`

nova	`openstack-nova-api`
	`openstack-nova-conductor`
	`openstack-nova-consoleauth`
	`openstack-nova-novncproxy`
	`openstack-nova-scheduler`
		`nova_metadata`
		`nova_api`
		`nova_conductor`
		`nova_vnc_proxy`
		`nova_consoleauth`
		`nova_api_cron`
		`nova_scheduler`
		`nova_placement`

panko		`panko_api`

OpenStack service	Systemd services	Docker containers
swift	`openstack-swift-account-auditor` `openstack-swift-account-reaper` `openstack-swift-account-replicator` `openstack-swift-account` `openstack-swift-container-auditor` `openstack-swift-container-replicator` `openstack-swift-container-updater` `openstack-swift-container` `openstack-swift-object-auditor` `openstack-swift-object-expirer` `openstack-swift-object-replicator` `openstack-swift-object-updater` `openstack-swift-object` `openstack-swift-proxy`	`swift_proxy` `swift_account_server` `swift_container_auditor` `swift_object_expirer` `swift_object_updater` `swift_container_replicator` `swift_account_auditor` `swift_object_replicator` `swift_container_server` `swift_rsync` `swift_account_reaper` `swift_account_replicator` `swift_object_auditor` `swift_object_server` `swift_container_update`

6.3. SYSTEMD LOG LOCATIONS VS CONTAINERIZED LOG LOCATIONS

The following table shows Systemd-based OpenStack logs and their equivalents for containers. All container-based log locations are available on the physical host and are mounted to the container.

OpenStack service	Systemd service logs	Docker container logs
aodh	`/var/log/aodh/`	`/var/log/containers/aodh/` `/var/log/containers/httpd/aodh-api/`

OpenStack service	Systemd service logs	Docker container logs
ceilometer	`/var/log/ceilometer/`	`/var/log/containers/ceilometer/`
cinder	`/var/log/cinder/`	`/var/log/containers/cinder/` `/var/log/containers/httpd/cinder-api/`
glance	`/var/log/glance/`	`/var/log/containers/glance/`
gnocchi	`/var/log/gnocchi/`	`/var/log/containers/gnocchi/` `/var/log/containers/httpd/gnocchi-api/`

heat	`/var/log/heat/`	`/var/log/containers/heat/`
		`/var/log/containers/httpd/heat-api/`
		`/var/log/containers/httpd/heat-api-cfn/`

horizon	`/var/log/horizon/`	`/var/log/containers/horizon/` `/var/log/containers/httpd/horizon/`
keystone	`/var/log/keystone/`	`/var/log/containers/keystone` `/var/log/containers/httpd/keystone/`

databases	`/var/log/mariadb/`
	`/var/log/mongodb/`
	`/var/log/mysqld.log`
		`/var/log/containers/mysql/`

neutron	`/var/log/neutron/`	`/var/log/containers/neutron/` `/var/log/containers/httpd/neutron-api/`

OpenStack service	Systemd service logs	Docker container logs
nova	**`/var/log/nova/`**	**`/var/log/containers/nova/`** **`/var/log/containers/httpd/nova-api/`** **`/var/log/containers/httpd/nova-placement/`**
panko		**`/var/log/containers/panko/`** **`/var/log/containers/httpd/panko-api/`**
rabbitmq	**`/var/log/rabbitmq/`**	**`/var/log/containers/rabbitmq/`**
redis	**`/var/log/redis/`**	**`/var/log/containers/redis/`**
swift	**`/var/log/swift/`**	**`/var/log/containers/swift/`**

6.4. SYSTEMD CONFIGURATION VS CONTAINERIZED CONFIGURATION

The following table shows Systemd-based OpenStack configuration and their equivalents for containers. All container-based configuration locations are available on the physical host, are mounted to the container, and are merged (via **`kolla`**) into the configuration within each respective container.

OpenStack service	Systemd service configuration	Docker container configuration
aodh	**`/etc/aodh/`**	**`/var/lib/config-data/puppet-generated/aodh/`**
ceilometer	**`/etc/ceilometer/`**	**`/var/lib/config-data/puppet-generated/ceilometer/etc/ceilometer/`**
cinder	**`/etc/cinder/`**	**`/var/lib/config-data/puppet-generated/cinder/etc/cinder/`**

OpenStack service	Systemd service configuration	Docker container configuration
glance	`/etc/glance/`	`/var/lib/config-data/puppet-generated/glance_api/etc/glance/`
gnocchi	`/etc/gnocchi/`	`/var/lib/config-data/puppet-generated/gnocchi/etc/gnocchi/`
haproxy	`/etc/haproxy/`	`/var/lib/config-data/puppet-generated/haproxy/etc/haproxy/`
heat	`/etc/heat/`	`/var/lib/config-data/puppet-generated/heat/etc/heat/` `/var/lib/config-data/puppet-generated/heat_api/etc/heat/` `/var/lib/config-data/puppet-generated/heat_api_cfn/etc/heat/`
horizon	`/etc/openstack-dashboard/`	`/var/lib/config-data/puppet-generated/horizon/etc/openstack-dashboard/`
keystone	`/etc/keystone/`	`/var/lib/config-data/puppet-generated/keystone/etc/keystone/`
databases	`/etc/my.cnf.d/` `/etc/my.cnf`	`/var/lib/config-data/puppet-generated/mysql/etc/my.cnf.d/`
neutron	`/etc/neutron/`	`/var/lib/config-data/puppet-generated/neutron/etc/neutron/`

OpenStack service	Systemd service configuration	Docker container configuration
nova	**`/etc/nova/`**	**`/var/lib/config-data/puppet-generated/nova/etc/nova/`** **`/var/lib/config-data/puppet-generated/nova_placement/etc/nova/`**
panko		**`/var/lib/config-data/puppet-generated/panko/etc/panko`**
rabbitmq	**`/etc/rabbitmq/`**	**`/var/lib/config-data/puppet-generated/rabbitmq/etc/rabbitmq/`**
redis	**`/etc/redis/`** **`/etc/redis.conf`**	**`/var/lib/config-data/puppet-generated/redis/etc/redis/`** **`/var/lib/config-data/puppet-generated/redis/etc/redis.conf`**
swift	**`/etc/swift/`**	**`/var/lib/config-data/puppet-generated/swift/etc/swift/`** **`/var/lib/config-data/puppet-generated/swift_ringbuilder/etc/swift/`**