



OpenShift Container Platform 4.5

Images

Creating and managing images and imagestreams in OpenShift Container Platform

OpenShift Container Platform 4.5 Images

Creating and managing images and imagestreams in OpenShift Container Platform

Legal Notice

Copyright © 2021 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This document provides instructions for creating and managing images and imagestreams in OpenShift Container Platform. It also provides instructions on using templates.

Table of Contents

CHAPTER 1. CONFIGURING THE CLUSTER SAMPLES OPERATOR	6
1.1. PREREQUISITES	6
1.2. UNDERSTANDING THE CLUSTER SAMPLES OPERATOR	6
1.2.1. Cluster Samples Operator's use of management state	7
1.2.1.1. Restricted network installation	7
1.2.1.2. Restricted network installation with initial network access	7
1.2.2. Cluster Samples Operator's tracking and error recovery of image stream imports	8
1.3. ADDITIONAL RESOURCES	8
1.4. CLUSTER SAMPLES OPERATOR CONFIGURATION PARAMETERS	8
1.4.1. Configuration restrictions	9
1.4.2. Conditions	10
1.5. ACCESSING THE CLUSTER SAMPLES OPERATOR CONFIGURATION	10
Additional resources	11
CHAPTER 2. USING THE CLUSTER SAMPLES OPERATOR WITH AN ALTERNATE REGISTRY	12
2.1. ABOUT THE MIRROR REGISTRY	12
2.1.1. Preparing the mirror host	12
2.1.2. Installing the CLI by downloading the binary	12
2.1.2.1. Installing the CLI on Linux	12
2.1.2.2. Installing the CLI on Windows	13
2.1.2.3. Installing the CLI on macOS	13
2.2. CONFIGURING CREDENTIALS THAT ALLOW IMAGES TO BE MIRRORED	14
2.3. MIRRORING THE OPENSIFT CONTAINER PLATFORM IMAGE REPOSITORY	16
2.4. USING CLUSTER SAMPLES OPERATOR IMAGE STREAMS WITH ALTERNATE OR MIRRORED REGISTRIES	19
CHAPTER 3. UNDERSTANDING CONTAINERS, IMAGES, AND IMAGE STREAMS	21
3.1. IMAGES	21
3.2. CONTAINERS	21
3.3. IMAGE REGISTRY	22
3.4. IMAGE REPOSITORY	22
3.5. IMAGE TAGS	22
3.6. IMAGE IDS	22
3.7. USING IMAGE STREAMS	22
3.8. IMAGE STREAM TAGS	23
3.9. IMAGE STREAM IMAGES	23
3.10. IMAGE STREAM TRIGGERS	24
3.11. ADDITIONAL RESOURCES	24
CHAPTER 4. CREATING IMAGES	25
4.1. LEARNING CONTAINER BEST PRACTICES	25
4.1.1. General container image guidelines	25
Reuse images	25
Maintain compatibility within tags	25
Avoid multiple processes	25
Use exec in wrapper scripts	25
Clean temporary files	26
Place instructions in the proper order	26
Mark important ports	27
Set environment variables	27
Avoid default passwords	27
Avoid sshd	27

Use volumes for persistent data	27
4.1.2. OpenShift Container Platform-specific guidelines	28
Enable images for source-to-image (S2I)	28
Support arbitrary user ids	28
Use services for inter-image communication	29
Provide common libraries	29
Use environment variables for configuration	29
Set image metadata	30
Clustering	30
Logging	30
Liveness and readiness probes	30
Templates	30
4.2. INCLUDING METADATA IN IMAGES	30
4.2.1. Defining image metadata	30
4.3. CREATING IMAGES FROM SOURCE CODE WITH SOURCE-TO-IMAGE	31
4.3.1. Understanding the source-to-image build process	32
4.3.2. How to write source-to-image scripts	32
4.4. ABOUT TESTING SOURCE-TO-IMAGE IMAGES	35
4.4.1. Understanding testing requirements	35
4.4.2. Generating scripts and tools	35
4.4.3. Testing locally	35
4.4.4. Basic testing workflow	36
4.4.5. Using OpenShift Container Platform for building the image	37
CHAPTER 5. MANAGING IMAGES	38
5.1. MANAGING IMAGES OVERVIEW	38
5.1.1. Images overview	38
5.2. TAGGING IMAGES	38
5.2.1. Image tags	38
5.2.2. Image tag conventions	38
5.2.3. Adding tags to image streams	39
5.2.4. Removing tags from image streams	40
5.2.5. Referencing images in imagestreams	40
5.3. IMAGE PULL POLICY	41
5.3.1. Image pull policy overview	41
5.4. USING IMAGE PULL SECRETS	42
5.4.1. Allowing pods to reference images across projects	42
5.4.2. Allowing pods to reference images from other secured registries	43
5.4.2.1. Pulling from private registries with delegated authentication	44
5.4.3. Updating the global cluster pull secret	44
CHAPTER 6. MANAGING IMAGE STREAMS	46
6.1. USING IMAGE STREAMS	46
6.2. CONFIGURING IMAGE STREAMS	47
6.3. IMAGE STREAM IMAGES	48
6.4. IMAGE STREAM TAGS	48
6.5. IMAGE STREAM CHANGE TRIGGERS	49
6.6. IMAGE STREAM MAPPING	49
6.7. WORKING WITH IMAGE STREAMS	52
6.7.1. Getting information about image streams	52
6.7.2. Adding tags to an image stream	53
6.7.3. Adding tags for an external image	54
6.7.4. Updating image stream tags	55

6.7.5. Removing image stream tags	55
6.7.6. Configuring periodic importing of image stream tags	55
6.8. IMPORTING IMAGES AND IMAGE STREAMS FROM PRIVATE REGISTRIES	56
6.8.1. Allowing pods to reference images from other secured registries	56
CHAPTER 7. USING IMAGE STREAMS WITH KUBERNETES RESOURCES	58
7.1. ENABLING IMAGE STREAMS WITH KUBERNETES RESOURCES	58
CHAPTER 8. TRIGGERING UPDATES ON IMAGE STREAM CHANGES	60
8.1. OPENSIFT CONTAINER PLATFORM RESOURCES	60
8.2. TRIGGERING KUBERNETES RESOURCES	60
8.3. SETTING THE IMAGE TRIGGER ON KUBERNETES RESOURCES	61
CHAPTER 9. IMAGE CONFIGURATION RESOURCES	62
9.1. IMAGE CONTROLLER CONFIGURATION PARAMETERS	62
9.2. CONFIGURING IMAGE SETTINGS	63
9.2.1. Adding specific registries	65
9.2.2. Blocking specific registries	68
9.2.3. Allowing insecure registries	70
9.2.4. Configuring additional trust stores for image registry access	71
9.2.5. Configuring image registry repository mirroring	72
CHAPTER 10. USING TEMPLATES	77
10.1. UNDERSTANDING TEMPLATES	77
10.2. UPLOADING A TEMPLATE	77
10.3. CREATING AN APPLICATION BY USING THE WEB CONSOLE	77
10.4. CREATING OBJECTS FROM TEMPLATES BY USING THE CLI	78
10.4.1. Adding labels	78
10.4.2. Listing parameters	78
10.4.3. Generating a list of objects	79
10.5. MODIFYING UPLOADED TEMPLATES	80
10.6. USING INSTANT APP AND QUICKSTART TEMPLATES	81
10.6.1. Quickstart templates	81
10.6.1.1. Web framework Quickstart templates	81
10.7. WRITING TEMPLATES	82
10.7.1. Writing the template description	82
10.7.2. Writing template labels	87
10.7.3. Writing template parameters	87
10.7.4. Writing the template object list	90
10.7.5. Marking a template as bindable	91
10.7.6. Exposing template object fields	91
10.7.7. Waiting for template readiness	93
10.7.8. Creating a template from existing objects	95
CHAPTER 11. USING RUBY ON RAILS	96
11.1. PREREQUISITES	96
11.2. SETTING UP THE DATABASE	96
11.3. WRITING YOUR APPLICATION	97
11.3.1. Creating a welcome page	98
11.3.2. Configuring application for OpenShift Container Platform	98
11.3.3. Storing your application in Git	99
11.4. DEPLOYING YOUR APPLICATION TO OPENSIFT CONTAINER PLATFORM	100
11.4.1. Creating the database service	100
11.4.2. Creating the frontend service	101

11.4.3. Creating a route for your application	102
CHAPTER 12. USING IMAGES	104
12.1. USING IMAGES OVERVIEW	104
12.2. CONFIGURING JENKINS IMAGES	104
12.2.1. Configuration and customization	104
12.2.1.1. OpenShift Container Platform OAuth authentication	105
12.2.1.2. Jenkins authentication	106
12.2.2. Jenkins environment variables	106
12.2.3. Providing Jenkins cross project access	110
12.2.4. Jenkins cross volume mount points	111
12.2.5. Customizing the Jenkins image through source-to-image	111
12.2.6. Configuring the Jenkins Kubernetes plug-in	112
12.2.7. Jenkins permissions	114
12.2.8. Creating a Jenkins service from a template	115
12.2.9. Using the Jenkins Kubernetes plug-in	116
12.2.10. Jenkins memory requirements	118
12.2.11. Additional Resources	118
12.3. JENKINS AGENT	118
12.3.1. Jenkins agent images	119
12.3.2. Jenkins agent environment variables	119
12.3.3. Jenkins agent memory requirements	121
12.3.4. Jenkins agent Gradle builds	121
12.3.5. Jenkins agent pod retention	122
12.4. SOURCE-TO-IMAGE	122
12.4.1. Source-to-image build process overview	123
12.4.2. Additional resources	123
12.5. CUSTOMIZING SOURCE-TO-IMAGE IMAGES	123
12.5.1. Invoking scripts embedded in an image	123

CHAPTER 1. CONFIGURING THE CLUSTER SAMPLES OPERATOR

The Cluster Samples Operator, which operates in the **openshift** namespace, installs and updates the Red Hat Enterprise Linux (RHEL)-based OpenShift Container Platform imagestreams and OpenShift Container Platform templates.

1.1. PREREQUISITES

- Deploy an OpenShift Container Platform cluster.

1.2. UNDERSTANDING THE CLUSTER SAMPLES OPERATOR

During installation, the Operator creates the default configuration object for itself and then creates the sample image streams and templates, including quickstart templates.



NOTE

To facilitate image stream imports from other registries that require credentials, a cluster administrator can create any additional secrets that contain the content of a Docker **config.json** file in the **openshift** namespace needed for image import.

The Cluster Samples Operator configuration is a cluster-wide resource, and the deployment is contained within the **openshift-cluster-samples-operator** namespace.

The image for the Cluster Samples Operator contains image stream and template definitions for the associated OpenShift Container Platform release. When each sample is created or updated, the Cluster Samples Operator includes an annotation that denotes the version of OpenShift Container Platform. The Operator uses this annotation to ensure that each sample matches the release version. Samples outside of its inventory are ignored, as are skipped samples. Modifications to any samples that are managed by the Operator, where that version annotation is modified or deleted, are reverted automatically.



NOTE

The Jenkins images are part of the image payload from installation and are tagged into the image streams directly.

The Cluster Samples Operator configuration resource includes a finalizer which cleans up the following upon deletion:

- Operator managed image streams.
- Operator managed templates.
- Operator generated configuration resources.
- Cluster status resources.

Upon deletion of the samples resource, the Cluster Samples Operator recreates the resource using the default configuration.

1.2.1. Cluster Samples Operator's use of management state

The Cluster Samples Operator is bootstrapped as **Managed** by default or if global proxy is configured. In the **Managed** state, the Cluster Samples Operator is actively managing its resources and keeping the component active in order to pull sample image streams and images from the registry and ensure that the requisite sample templates are installed.

Certain circumstances result in the Cluster Samples Operator bootstrapping itself as **Removed** including:

- If the Cluster Samples Operator cannot reach registry.redhat.io after three minutes on initial start-up after a clean installation.
- If the Cluster Samples Operator detects it is on an IPv6 network.

However, if the Cluster Samples Operator also detects an OpenShift Container Platform global proxy is configured, it bypasses those checks.



IMPORTANT

IPv6 installations are not currently supported by registry.redhat.io. The Cluster Samples Operator pulls most of the sample image streams and images from registry.redhat.io.

1.2.1.1. Restricted network installation

Boostrapping as **Removed** when unable to access registry.redhat.io facilitates restricted network installations when the network restriction is already in place. Bootstrapping as **Removed** when network access is restricted allows the cluster administrator more time to decide if samples are desired, because the Cluster Samples Operator does not submit alerts that sample image stream imports are failing when the management state is set to **Removed**. When the Cluster Samples Operator comes up as **Managed** and attempts to install sample image streams, it starts alerting two hours after initial installation if there are failing imports.

1.2.1.2. Restricted network installation with initial network access

Conversely, if a cluster that is intended to be a restricted network or disconnected cluster is first installed while network access exists, the Cluster Samples Operator installs the content from registry.redhat.io since it can access it. If you want the Cluster Samples Operator to still bootstrap as **Removed** in order to defer samples installation until you have decided which samples are desired, set up image mirrors, and so on, then follow the instructions for using the Samples Operator with an alternate registry and customizing nodes, both linked in the additional resources section, to override the Cluster Samples Operator default configuration and initially come up as **Removed**.

You must put the following additional YAML file in the **openshift** directory created by **openshift-install create manifest**:

Example Cluster Samples Operator YAML file with **managementState: Removed**

```
apiVersion: samples.operator.openshift.io/v1
kind: Config
metadata:
  name: cluster
spec:
```

```
architectures:
- x86_64
managementState: Removed
```

1.2.2. Cluster Samples Operator's tracking and error recovery of image stream imports

After creation or update of a samples image stream, the Cluster Samples Operator monitors the progress of each image stream tag's image import.

If an import fails, the Cluster Samples Operator retries the import through the image stream image import API, which is the same API used by the **oc import-image** command, approximately every 15 minutes until it sees the import succeed, or if the Cluster Samples Operator's configuration is changed such that either the image stream is added to the **skippedImagestreams** list, or the management state is changed to **Removed**.


1.3. ADDITIONAL RESOURCES

- If the Cluster Samples Operator is removed during installation, you can [use the Cluster Samples Operator with an alternate registry](#).
- To ensure the Cluster Samples Operator bootstraps as **Removed** in a restricted network installation with initial network access in order to defer samples installation until you have decided which samples are desired, follow the instructions for [using the Cluster Samples Operator with an alternate registry](#) and [Customizing nodes](#), to override the Cluster Samples Operator default configuration and initially come up as **Removed**.

1.4. CLUSTER SAMPLES OPERATOR CONFIGURATION PARAMETERS

The samples resource offers the following configuration fields:

Parameter	Description
managementState	<p>Managed: The Cluster Samples Operator updates the samples as the configuration dictates.</p> <p>Unmanaged: The Cluster Samples Operator ignores updates to its configuration resource object and any image streams or templates in the openshift namespace.</p> <p>Removed: The Cluster Samples Operator removes the set of Managed image streams and templates in the openshift namespace. It ignores new samples created by the cluster administrator or any samples in the skipped lists. After the removals are complete, the Cluster Samples Operator works like it is in the Unmanaged state and ignores any watch events on the sample resources, image streams, or templates.</p>

Parameter	Description
samplesRegistry	<p>Allows you to specify which registry is accessed by image streams for their image content. samplesRegistry defaults to registry.redhat.io for OpenShift Container Platform.</p> <div style="display: flex; align-items: flex-start;"> <div style="flex: 1;">  </div> <div style="flex: 2;"> <p>NOTE</p> <p>Creation or update of RHEL content does not commence if the secret for pull access is not in place when either Samples Registry is not explicitly set, leaving an empty string, or when it is set to registry.redhat.io. In both cases, image imports work off of registry.redhat.io, which requires credentials.</p> <p>Creation or update of RHEL content is not gated by the existence of the pull secret if the Samples Registry is overridden to a value other than the empty string or registry.redhat.io.</p> </div> </div>
architectures	Placeholder to choose an architecture type.
skippedImagestreams	Image streams that are in the Cluster Samples Operator's inventory but that the cluster administrator wants the Operator to ignore or not manage. You can add a list of image stream names to this parameter. For example, ["httpd","perl"] .
skippedTemplates	Templates that are in the Cluster Samples Operator's inventory, but that the cluster administrator wants the Operator to ignore or not manage.

Secret, image stream, and template watch events can come in before the initial samples resource object is created, the Cluster Samples Operator detects and re-queues the event.

1.4.1. Configuration restrictions

When the Cluster Samples Operator starts supporting multiple architectures, the architecture list is not allowed to be changed while in the **Managed** state.

In order to change the architectures values, a cluster administrator must:

- Mark the **Management State** as **Removed**, saving the change.
- In a subsequent change, edit the architecture and change the **Management State** back to **Managed**.

The Cluster Samples Operator still processes secrets while in **Removed** state. You can create the secret before switching to **Removed**, while in **Removed** before switching to **Managed**, or after switching to **Managed** state. There are delays in creating the samples until the secret event is processed if you create the secret after switching to **Managed**. This helps facilitate the changing of the registry, where you choose to remove all the samples before switching to insure a clean slate. Removing all samples before switching is not required.

1.4.2. Conditions

The samples resource maintains the following conditions in its status:

Condition	Description
SamplesExists	Indicates the samples are created in the openshift namespace.
ImageChangesInProgress	<p>True when image streams are created or updated, but not all of the tag spec generations and tag status generations match.</p> <p>False when all of the generations match, or unrecoverable errors occurred during import, the last seen error is in the message field. The list of pending image streams is in the reason field.</p>
ConfigurationValid	True or False based on whether any of the restricted changes noted previously are submitted.
RemovePending	Indicator that there is a Management State: Removed setting pending, but the Cluster Samples Operator is waiting for the deletions to complete.
ImportImageErrorsExist	<p>Indicator of which image streams had errors during the image import phase for one of their tags.</p> <p>True when an error has occurred. The list of image streams with an error is in the reason field. The details of each error reported are in the message field.</p>
MigrationInProgress	True when the Cluster Samples Operator detects that the version is different than the Cluster Samples Operator version with which the current samples set are installed.

1.5. ACCESSING THE CLUSTER SAMPLES OPERATOR CONFIGURATION

You can configure the Cluster Samples Operator by editing the file with the provided parameters.

Prerequisites

- Install the OpenShift CLI (**oc**).

Procedure

- Access the Cluster Samples Operator configuration:

```
$ oc edit configs.samples.operator.openshift.io/cluster -o yaml
```

The Cluster Samples Operator configuration resembles the following example:

```
apiVersion: samples.operator.openshift.io/v1
kind: Config
...
```

Additional resources

For more information about configuring credentials, see [Using image pull secrets](#).

CHAPTER 2. USING THE CLUSTER SAMPLES OPERATOR WITH AN ALTERNATE REGISTRY

You can use the Cluster Samples Operator with an alternate registry by first creating a mirror registry.



IMPORTANT

You must have access to the internet to obtain the necessary container images. In this procedure, you place the mirror registry on a mirror host that has access to both your network and the internet.

2.1. ABOUT THE MIRROR REGISTRY

You can mirror the images that are required for OpenShift Container Platform installation and subsequent product updates to a mirror registry. These actions use the same process. The release image, which contains the description of the content, and the images it references are all mirrored. In addition, the Operator catalog source image and the images that it references must be mirrored for each Operator that you use. After you mirror the content, you configure each cluster to retrieve this content from your mirror registry.

The mirror registry can be any container registry that supports [Docker v2-2](#). All major cloud provider registries, as well as Red Hat Quay, Artifactory, and others, have the necessary support. Using one of these registries ensures that OpenShift Container Platform can verify the integrity of each image in disconnected environments.

The mirror registry must be reachable by every machine in the clusters that you provision. If the registry is unreachable installation, updating, or normal operations such as workload relocation might fail. For that reason, you must run mirror registries in a highly available way, and the mirror registries must at least match the production availability of your OpenShift Container Platform clusters.

When you populate a mirror registry with OpenShift Container Platform images, you can follow two scenarios. If you have a host that can access both the internet and your mirror registry, but not your cluster nodes, you can directly mirror the content from that machine. This process is referred to as *connected mirroring*. If you have no such host, you must mirror the images to a file system and then bring that host or removable media into your restricted environment. This process is referred to as *disconnected mirroring*.

2.1.1. Preparing the mirror host

Before you create the mirror registry, you must prepare the mirror host.

2.1.2. Installing the CLI by downloading the binary

You can install the OpenShift CLI (**oc**) in order to interact with OpenShift Container Platform from a command-line interface. You can install **oc** on Linux, Windows, or macOS.



IMPORTANT

If you installed an earlier version of **oc**, you cannot use it to complete all of the commands in OpenShift Container Platform 4.5. Download and install the new version of **oc**.

2.1.2.1. Installing the CLI on Linux

You can install the OpenShift CLI (**oc**) binary on Linux by using the following procedure.

Procedure

1. Navigate to the [Infrastructure Provider](#) page on the Red Hat OpenShift Cluster Manager site.
2. Select your infrastructure provider, and, if applicable, your installation type.
3. In the **Command line interface** section, select **Linux** from the drop-down menu and click **Download command-line tools**.
4. Unpack the archive:

```
$ tar xvzf <file>
```

5. Place the **oc** binary in a directory that is on your **PATH**.
To check your **PATH**, execute the following command:

```
$ echo $PATH
```

After you install the CLI, it is available using the **oc** command:

```
$ oc <command>
```

2.1.2.2. Installing the CLI on Windows

You can install the OpenShift CLI (**oc**) binary on Windows by using the following procedure.

Procedure

1. Navigate to the [Infrastructure Provider](#) page on the Red Hat OpenShift Cluster Manager site.
2. Select your infrastructure provider, and, if applicable, your installation type.
3. In the **Command line interface** section, select **Windows** from the drop-down menu and click **Download command-line tools**.
4. Unzip the archive with a ZIP program.
5. Move the **oc** binary to a directory that is on your **PATH**.
To check your **PATH**, open the command prompt and execute the following command:

```
C:\> path
```

After you install the CLI, it is available using the **oc** command:

```
C:\> oc <command>
```

2.1.2.3. Installing the CLI on macOS

You can install the OpenShift CLI (**oc**) binary on macOS by using the following procedure.

Procedure

1. Navigate to the [Infrastructure Provider](#) page on the Red Hat OpenShift Cluster Manager site.
2. Select your infrastructure provider, and, if applicable, your installation type.
3. In the **Command line interface** section, select **MacOS** from the drop-down menu and click **Download command-line tools**.
4. Unpack and unzip the archive.
5. Move the **oc** binary to a directory on your PATH.
To check your **PATH**, open a terminal and execute the following command:

```
$ echo $PATH
```

After you install the CLI, it is available using the **oc** command:

```
$ oc <command>
```

2.2. CONFIGURING CREDENTIALS THAT ALLOW IMAGES TO BE MIRRORED

Create a container image registry credentials file that allows mirroring images from Red Hat to your mirror.

Prerequisites

- You configured a mirror registry to use in your restricted network.

Procedure

Complete the following steps on the installation host:

1. Download your **registry.redhat.io** pull secret from the [Pull Secret](#) page on the Red Hat OpenShift Cluster Manager site and save it to a **.json** file.
2. Generate the base64-encoded user name and password or token for your mirror registry:

```
$ echo -n '<user_name>:<password>' | base64 -w0 1  
BGVtbYk3ZHAtdXs=
```

- 1** For **<user_name>** and **<password>**, specify the user name and password that you configured for your registry.

3. Make a copy of your pull secret in JSON format:

```
$ cat ./pull-secret.text | jq . > <path>/<pull-secret-file> 1
```

- 1** Specify the path to the folder to store the pull secret in and a name for the JSON file that you create.

The contents of the file resemble the following example:

```
{
  "auths": {
    "cloud.openshift.com": {
      "auth": "b3BlbnNo...",
      "email": "you@example.com"
    },
    "quay.io": {
      "auth": "b3BlbnNo...",
      "email": "you@example.com"
    },
    "registry.connect.redhat.com": {
      "auth": "NTE3Njg5Nj...",
      "email": "you@example.com"
    },
    "registry.redhat.io": {
      "auth": "NTE3Njg5Nj...",
      "email": "you@example.com"
    }
  }
}
```

4. Edit the new file and add a section that describes your registry to it:

```
"auths": {
  "<mirror_registry>": { 1
    "auth": "<credentials>", 2
    "email": "you@example.com"
  },
}
```

- 1** For **<mirror_registry>**, specify the registry domain name, and optionally the port, that your mirror registry uses to serve content. For example, **registry.example.com** or **registry.example.com:5000**
- 2** For **<credentials>**, specify the base64-encoded user name and password for the mirror registry.

The file resembles the following example:

```
{
  "auths": {
    "<mirror_registry>": {
      "auth": "<credentials>",
      "email": "you@example.com"
    },
    "cloud.openshift.com": {
      "auth": "b3BlbnNo...",
      "email": "you@example.com"
    },
    "quay.io": {
      "auth": "b3BlbnNo...",
      "email": "you@example.com"
    },
  }
}
```

```

"registry.connect.redhat.com": {
  "auth": "NTE3Njg5Nj...",
  "email": "you@example.com"
},
"registry.redhat.io": {
  "auth": "NTE3Njg5Nj...",
  "email": "you@example.com"
}
}
}
}

```

2.3. MIRRORING THE OPENSIFT CONTAINER PLATFORM IMAGE REPOSITORY

Mirror the OpenShift Container Platform image repository to your registry to use during cluster installation or upgrade.

Prerequisites

- Your mirror host has access to the Internet.
- You configured a mirror registry to use in your restricted network and can access the certificate and credentials that you configured.
- You downloaded the pull secret from the [Pull Secret](#) page on the Red Hat OpenShift Cluster Manager site and modified it to include authentication to your mirror repository.
- If you use self-signed certificates that do not set a Subject Alternative Name, you must precede the **oc** commands in this procedure with **GODEBUG=x509ignoreCN=0**. If you do not set this variable, the **oc** commands will fail with the following error:

```
x509: certificate relies on legacy Common Name field, use SANs or temporarily enable
Common Name matching with GODEBUG=x509ignoreCN=0
```

Procedure

Complete the following steps on the mirror host:

1. Review the [OpenShift Container Platform downloads page](#) to determine the version of OpenShift Container Platform that you want to install and determine the corresponding tag on the [Repository Tags](#) page.
2. Set the required environment variables:
 - a. Export the release version:

```
$ OCP_RELEASE=<release_version>
```

For **<release_version>**, specify the tag that corresponds to the version of OpenShift Container Platform to install, such as **4.5.4**.

- b. Export the local registry name and host port:

```
$ LOCAL_REGISTRY='<local_registry_host_name>:<local_registry_host_port>'
```

For **<local_registry_host_name>**, specify the registry domain name for your mirror repository, and for **<local_registry_host_port>**, specify the port that it serves content on.

- c. Export the local repository name:

```
$ LOCAL_REPOSITORY='<local_repository_name>'
```

For **<local_repository_name>**, specify the name of the repository to create in your registry, such as **ocp4/openshift4**.

- d. Export the name of the repository to mirror:

```
$ PRODUCT_REPO='openshift-release-dev'
```

For a production release, you must specify **openshift-release-dev**.

- e. Export the path to your registry pull secret:

```
$ LOCAL_SECRET_JSON='<path_to_pull_secret>'
```

For **<path_to_pull_secret>**, specify the absolute path to and file name of the pull secret for your mirror registry that you created.

- f. Export the release mirror:

```
$ RELEASE_NAME="ocp-release"
```

For a production release, you must specify **ocp-release**.

- g. Export the type of architecture for your server, such as **x86_64**:

```
$ ARCHITECTURE=<server_architecture>
```

- h. Export the path to the directory to host the mirrored images:

```
$ REMOVABLE_MEDIA_PATH=<path> 1
```

- 1** Specify the full path, including the initial forward slash (/) character.

3. Mirror the version images to the internal container registry:

- If your mirror host does not have Internet access, take the following actions:
 - i. Connect the removable media to a system that is connected to the Internet.
 - ii. Review the images and configuration manifests to mirror:

```
$ oc adm release mirror -a ${LOCAL_SECRET_JSON} \
  --from=quay.io/${PRODUCT_REPO}/${RELEASE_NAME}:${OCP_RELEASE}-
  ${ARCHITECTURE} \
  --to=${LOCAL_REGISTRY}/${LOCAL_REPOSITORY} \
  --to-release-
  image=${LOCAL_REGISTRY}/${LOCAL_REPOSITORY}:${OCP_RELEASE}-
  ${ARCHITECTURE} --dry-run
```

- iii. Record the entire **imageContentSources** section from the output of the previous command. The information about your mirrors is unique to your mirrored repository, and you must add the **imageContentSources** section to the **install-config.yaml** file during installation.
- iv. Mirror the images to a directory on the removable media:

```
$ oc adm release mirror -a ${LOCAL_SECRET_JSON} --to-dir=${REMOVABLE_MEDIA_PATH}/mirror quay.io/${PRODUCT_REPO}/${RELEASE_NAME}:${OCP_RELEASE}-${ARCHITECTURE}
```

- v. Take the media to the restricted network environment and upload the images to the local container registry.

```
$ oc image mirror -a ${LOCAL_SECRET_JSON} --from-dir=${REMOVABLE_MEDIA_PATH}/mirror "file://openshift/release:${OCP_RELEASE}*" ${LOCAL_REGISTRY}/${LOCAL_REPOSITORY} 1
```

- 1** For **REMOVABLE_MEDIA_PATH**, you must use the same path that you specified when you mirrored the images.

- If the local container registry is connected to the mirror host, take the following actions:
 - i. Directly push the release images to the local registry by using following command:

```
$ oc adm release mirror -a ${LOCAL_SECRET_JSON} \ --from=quay.io/${PRODUCT_REPO}/${RELEASE_NAME}:${OCP_RELEASE}-${ARCHITECTURE} \ --to=${LOCAL_REGISTRY}/${LOCAL_REPOSITORY} \ --to-release-image=${LOCAL_REGISTRY}/${LOCAL_REPOSITORY}:${OCP_RELEASE}-${ARCHITECTURE}
```

This command pulls the release information as a digest, and its output includes the **imageContentSources** data that you require when you install your cluster.

- ii. Record the entire **imageContentSources** section from the output of the previous command. The information about your mirrors is unique to your mirrored repository, and you must add the **imageContentSources** section to the **install-config.yaml** file during installation.



NOTE

The image name gets patched to Quay.io during the mirroring process, and the podman images will show Quay.io in the registry on the bootstrap virtual machine.

4. To create the installation program that is based on the content that you mirrored, extract it and pin it to the release:
 - If your mirror host does not have Internet access, run the following command:

```
$ oc adm release extract -a ${LOCAL_SECRET_JSON} --command=openshift-install
"${LOCAL_REGISTRY}/${LOCAL_REPOSITORY}:${OCP_RELEASE}"
```

- If the local container registry is connected to the mirror host, run the following command:

```
$ oc adm release extract -a ${LOCAL_SECRET_JSON} --command=openshift-install
"${LOCAL_REGISTRY}/${LOCAL_REPOSITORY}:${OCP_RELEASE}-
${ARCHITECTURE}"
```



IMPORTANT

To ensure that you use the correct images for the version of OpenShift Container Platform that you selected, you must extract the installation program from the mirrored content.

You must perform this step on a machine with an active Internet connection.

If you are in a disconnected environment, use the **--image** flag as part of `must-gather` and point to the payload image.

2.4. USING CLUSTER SAMPLES OPERATOR IMAGE STREAMS WITH ALTERNATE OR MIRRORED REGISTRIES

Most image streams in the **openshift** namespace managed by the Cluster Samples Operator point to images located in the Red Hat registry at registry.redhat.io.



IMPORTANT

The **jenkins**, **jenkins-agent-maven**, and **jenkins-agent-nodejs** image streams come from the install payload and are managed by the Samples Operator.

Setting the **samplesRegistry** field in the Sample Operator configuration file to registry.redhat.io is redundant because it is already directed to registry.redhat.io for everything but Jenkins images and image streams.



NOTE

The **cli**, **installer**, **must-gather**, and **tests** image streams, while part of the install payload, are not managed by the Cluster Samples Operator. These are not addressed in this procedure.

Prerequisites

- Access to the cluster as a user with the **cluster-admin** role.
- Create a pull secret for your mirror registry.

Procedure

1. Access the images of a specific image stream to mirror, for example:

```
$ oc get is <imagestream> -n openshift -o json | jq .spec.tags[].from.name | grep
registry.redhat.io
```

-
- 2. Mirror images from registry.redhat.io associated with any image streams you need

```
$ oc image mirror registry.redhat.io/rhsc/ruby-25-rhel7:latest ${MIRROR_ADDR}/rhsc/ruby-25-rhel7:latest
```

- 3. Create the cluster's image configuration object:

```
$ oc create configmap registry-config --from-file=${MIRROR_ADDR_HOSTNAME}..5000=$path/ca.crt -n openshift-config
```

- 4. Add the required trusted CAs for the mirror in the cluster's image configuration object:

```
$ oc patch image.config.openshift.io/cluster --patch '{"spec":{"additionalTrustedCA":{"name":"registry-config"}}}' --type=merge
```

- 5. Update the **samplesRegistry** field in the Cluster Samples Operator configuration object to contain the **hostname** portion of the mirror location defined in the mirror configuration:

```
$ oc edit configs.samples.operator.openshift.io -n openshift-cluster-samples-operator
```



NOTE

This is required because the image stream import process does not use the mirror or search mechanism at this time.

- 6. Add any image streams that are not mirrored into the **skippedImagestreams** field of the Cluster Samples Operator configuration object. Or if you do not want to support any of the sample image streams, set the Cluster Samples Operator to **Removed** in the Cluster Samples Operator configuration object.



NOTE

Any unmirrored image streams that are not skipped, or if the Samples Operator is not changed to **Removed**, will result in the Samples Operator reporting a **Degraded** status two hours after the image stream imports start failing.

Many of the templates in the **openshift** namespace reference the image streams. So using **Removed** to purge both the image streams and templates will eliminate the possibility of attempts to use them if they are not functional because of any missing image streams.

CHAPTER 3. UNDERSTANDING CONTAINERS, IMAGES, AND IMAGE STREAMS

Containers, images, and image streams are important concepts to understand when you set out to create and manage containerized software. An image holds a set of software that is ready to run, while a container is a running instance of a container image. An image stream provides a way of storing different versions of the same basic image. Those different versions are represented by different tags on the same image name.

3.1. IMAGES

Containers in OpenShift Container Platform are based on OCI- or Docker-formatted container *images*. An image is a binary that includes all of the requirements for running a single container, as well as metadata describing its needs and capabilities.

You can think of it as a packaging technology. Containers only have access to resources defined in the image unless you give the container additional access when creating it. By deploying the same image in multiple containers across multiple hosts and load balancing between them, OpenShift Container Platform can provide redundancy and horizontal scaling for a service packaged into an image.

You can use the [podman](#) or **docker** CLI directly to build images, but OpenShift Container Platform also supplies builder images that assist with creating new images by adding your code or configuration to existing images.

Because applications develop over time, a single image name can actually refer to many different versions of the same image. Each different image is referred to uniquely by its hash, a long hexadecimal number such as **fd44297e2ddb050ec4f...**, which is usually shortened to 12 characters, such as **fd44297e2ddb**.

3.2. CONTAINERS

The basic units of OpenShift Container Platform applications are called containers. [Linux container technologies](#) are lightweight mechanisms for isolating running processes so that they are limited to interacting with only their designated resources. The word container is defined as a specific running or paused instance of a container image.

Many application instances can be running in containers on a single host without visibility into each others' processes, files, network, and so on. Typically, each container provides a single service, often called a micro-service, such as a web server or a database, though containers can be used for arbitrary workloads.

The Linux kernel has been incorporating capabilities for container technologies for years. The Docker project developed a convenient management interface for Linux containers on a host. More recently, the [Open Container Initiative](#) has developed open standards for container formats and container runtimes. OpenShift Container Platform and Kubernetes add the ability to orchestrate OCI- and Docker-formatted containers across multi-host installations.

Though you do not directly interact with container runtimes when using OpenShift Container Platform, understanding their capabilities and terminology is important for understanding their role in OpenShift Container Platform and how your applications function inside of containers.

Tools such as [podman](#) can be used to replace **docker** command-line tools for running and managing containers directly. Using **podman**, you can experiment with containers separately from OpenShift Container Platform.

3.3. IMAGE REGISTRY

An image registry is a content server that can store and serve container images. For example:

```
registry.redhat.io
```

A registry contains a collection of one or more image repositories, which contain one or more tagged images. Red Hat provides a registry at **registry.redhat.io** for subscribers. OpenShift Container Platform can also supply its own internal registry for managing custom container images.

3.4. IMAGE REPOSITORY

An image repository is a collection of related container images and tags identifying them. For example, the OpenShift Container Platform Jenkins images are in the repository:

```
docker.io/openshift/jenkins-2-centos7
```

3.5. IMAGE TAGS

An image tag is a label applied to a container image in a repository that distinguishes a specific image from other images in an image stream. Typically, the tag represents a version number of some sort. For example, here **:v3.11.59-2** is the tag:

```
registry.access.redhat.com/openshift3/jenkins-2-rhel7:v3.11.59-2
```

You can add additional tags to an image. For example, an image might be assigned the tags **:v3.11.59-2** and **:latest**.

OpenShift Container Platform provides the **oc tag** command, which is similar to the **docker tag** command, but operates on image streams instead of directly on images.

3.6. IMAGE IDS

An image ID is a SHA (Secure Hash Algorithm) code that can be used to pull an image. A SHA image ID cannot change. A specific SHA identifier always references the exact same container image content. For example:

```
docker.io/openshift/jenkins-2-centos7@sha256:ab312bda324
```

3.7. USING IMAGE STREAMS

An image stream and its associated tags provide an abstraction for referencing container images from within OpenShift Container Platform. The image stream and its tags allow you to see what images are available and ensure that you are using the specific image you need even if the image in the repository changes.

Image streams do not contain actual image data, but present a single virtual view of related images, similar to an image repository.

You can configure builds and deployments to watch an image stream for notifications when new images are added and react by performing a build or deployment, respectively.

For example, if a deployment is using a certain image and a new version of that image is created, a deployment could be automatically performed to pick up the new version of the image.

However, if the image stream tag used by the deployment or build is not updated, then even if the container image in the container image registry is updated, the build or deployment continues using the previous, presumably known good image.

The source images can be stored in any of the following:

- OpenShift Container Platform’s integrated registry.
- An external registry, for example registry.redhat.io or Quay.io.
- Other image streams in the OpenShift Container Platform cluster.

When you define an object that references an image stream tag, such as a build or deployment configuration, you point to an image stream tag and not the repository. When you build or deploy your application, OpenShift Container Platform queries the repository using the image stream tag to locate the associated ID of the image and uses that exact image.

The image stream metadata is stored in the etcd instance along with other cluster information.

Using image streams has several significant benefits:

- You can tag, rollback a tag, and quickly deal with images, without having to re-push using the command line.
- You can trigger builds and deployments when a new image is pushed to the registry. Also, OpenShift Container Platform has generic triggers for other resources, such as Kubernetes objects.
- You can mark a tag for periodic re-import. If the source image has changed, that change is picked up and reflected in the image stream, which triggers the build or deployment flow, depending upon the build or deployment configuration.
- You can share images using fine-grained access control and quickly distribute images across your teams.
- If the source image changes, the image stream tag still points to a known-good version of the image, ensuring that your application do not break unexpectedly.
- You can configure security around who can view and use the images through permissions on the image stream objects.
- Users that lack permission to read or list images on the cluster level can still retrieve the images tagged in a project using image streams.

3.8. IMAGE STREAM TAGS

An image stream tag is a named pointer to an image in an image stream. An image stream tag is similar to a container image tag.

3.9. IMAGE STREAM IMAGES

An image stream image allows you to retrieve a specific container image from a particular image stream where it is tagged. An image stream image is an API resource object that pulls together some metadata about a particular image SHA identifier.

3.10. IMAGE STREAM TRIGGERS

An image stream trigger causes a specific action when an image stream tag changes. For example, importing can cause the value of the tag to change, which causes a trigger to fire when there are deployments, builds, or other resources listening for those.

3.11. ADDITIONAL RESOURCES

- For more information on using image streams, see [Managing image streams](#).

CHAPTER 4. CREATING IMAGES

Learn how to create your own container images, based on pre-built images that are ready to help you. The process includes learning best practices for writing images, defining metadata for images, testing images, and using a custom builder workflow to create images to use with OpenShift Container Platform. After you create an image, you can push it to the internal registry.

4.1. LEARNING CONTAINER BEST PRACTICES

When creating container images to run on OpenShift Container Platform there are a number of best practices to consider as an image author to ensure a good experience for consumers of those images. Because images are intended to be immutable and used as-is, the following guidelines help ensure that your images are highly consumable and easy to use on OpenShift Container Platform.

4.1.1. General container image guidelines

The following guidelines apply when creating a container image in general, and are independent of whether the images are used on OpenShift Container Platform.

Reuse images

Wherever possible, base your image on an appropriate upstream image using the **FROM** statement. This ensures your image can easily pick up security fixes from an upstream image when it is updated, rather than you having to update your dependencies directly.

In addition, use tags in the **FROM** instruction, for example, **rhel:rhel7**, to make it clear to users exactly which version of an image your image is based on. Using a tag other than **latest** ensures your image is not subjected to breaking changes that might go into the **latest** version of an upstream image.

Maintain compatibility within tags

When tagging your own images, try to maintain backwards compatibility within a tag. For example, if you provide an image named **foo** and it currently includes version **1.0**, you might provide a tag of **foo:v1**. When you update the image, as long as it continues to be compatible with the original image, you can continue to tag the new image **foo:v1**, and downstream consumers of this tag are able to get updates without being broken.

If you later release an incompatible update, then switch to a new tag, for example **foo:v2**. This allows downstream consumers to move up to the new version at will, but not be inadvertently broken by the new incompatible image. Any downstream consumer using **foo:latest** takes on the risk of any incompatible changes being introduced.

Avoid multiple processes

Do not start multiple services, such as a database and **SSHD**, inside one container. This is not necessary because containers are lightweight and can be easily linked together for orchestrating multiple processes. OpenShift Container Platform allows you to easily colocate and co-manage related images by grouping them into a single pod.

This colocation ensures the containers share a network namespace and storage for communication. Updates are also less disruptive as each image can be updated less frequently and independently. Signal handling flows are also clearer with a single process as you do not have to manage routing signals to spawned processes.

Use **exec** in wrapper scripts

Many images use wrapper scripts to do some setup before starting a process for the software being run. If your image uses such a script, that script uses **exec** so that the script's process is replaced by your software. If you do not use **exec**, then signals sent by your container runtime go to your wrapper script

instead of your software's process. This is not what you want.

If you have a wrapper script that starts a process for some server. You start your container, for example, using **podman run -i**, which runs the wrapper script, which in turn starts your process. If you want to close your container with **CTRL+C**. If your wrapper script used **exec** to start the server process, **podman** sends SIGINT to the server process, and everything works as you expect. If you did not use **exec** in your wrapper script, **podman** sends SIGINT to the process for the wrapper script and your process keeps running like nothing happened.

Also note that your process runs as **PID 1** when running in a container. This means that if your main process terminates, the entire container is stopped, canceling any child processes you launched from your **PID 1** process.

Clean temporary files

Remove all temporary files you create during the build process. This also includes any files added with the **ADD** command. For example, run the **yum clean** command after performing **yum install** operations.

You can prevent the **yum** cache from ending up in an image layer by creating your **RUN** statement as follows:

```
RUN yum -y install mypackage && yum -y install myotherpackage && yum clean all -y
```

Note that if you instead write:

```
RUN yum -y install mypackage  
RUN yum -y install myotherpackage && yum clean all -y
```

Then the first **yum** invocation leaves extra files in that layer, and these files cannot be removed when the **yum clean** operation is run later. The extra files are not visible in the final image, but they are present in the underlying layers.

The current container build process does not allow a command run in a later layer to shrink the space used by the image when something was removed in an earlier layer. However, this may change in the future. This means that if you perform an **rm** command in a later layer, although the files are hidden it does not reduce the overall size of the image to be downloaded. Therefore, as with the **yum clean** example, it is best to remove files in the same command that created them, where possible, so they do not end up written to a layer.

In addition, performing multiple commands in a single **RUN** statement reduces the number of layers in your image, which improves download and extraction time.

Place instructions in the proper order

The container builder reads the **Dockerfile** and runs the instructions from top to bottom. Every instruction that is successfully executed creates a layer which can be reused the next time this or another image is built. It is very important to place instructions that rarely change at the top of your **Dockerfile**. Doing so ensures the next builds of the same image are very fast because the cache is not invalidated by upper layer changes.

For example, if you are working on a **Dockerfile** that contains an **ADD** command to install a file you are iterating on, and a **RUN** command to **yum install** a package, it is best to put the **ADD** command last:

```
FROM foo  
RUN yum -y install mypackage && yum clean all -y  
ADD myfile /test/myfile
```

This way each time you edit **myfile** and rerun **podman build** or **docker build**, the system reuses the cached layer for the **yum** command and only generates the new layer for the **ADD** operation.

If instead you wrote the **Dockerfile** as:

```
FROM foo
ADD myfile /test/myfile
RUN yum -y install mypackage && yum clean all -y
```

Then each time you changed **myfile** and reran **podman build** or **docker build**, the **ADD** operation would invalidate the **RUN** layer cache, so the **yum** operation must be rerun as well.

Mark important ports

The EXPOSE instruction makes a port in the container available to the host system and other containers. While it is possible to specify that a port should be exposed with a **podman run** invocation, using the EXPOSE instruction in a **Dockerfile** makes it easier for both humans and software to use your image by explicitly declaring the ports your software needs to run:

- Exposed ports show up under **podman ps** associated with containers created from your image.
- Exposed ports are present in the metadata for your image returned by **podman inspect**.
- Exposed ports are linked when you link one container to another.

Set environment variables

It is good practice to set environment variables with the **ENV** instruction. One example is to set the version of your project. This makes it easy for people to find the version without looking at the **Dockerfile**. Another example is advertising a path on the system that could be used by another process, such as **JAVA_HOME**.

Avoid default passwords

Avoid setting default passwords. Many people extend the image and forget to remove or change the default password. This can lead to security issues if a user in production is assigned a well-known password. Passwords are configurable using an environment variable instead.

If you do choose to set a default password, ensure that an appropriate warning message is displayed when the container is started. The message should inform the user of the value of the default password and explain how to change it, such as what environment variable to set.

Avoid sshd

It is best to avoid running **sshd** in your image. You can use the **podman exec** or **docker exec** command to access containers that are running on the local host. Alternatively, you can use the **oc exec** command or the **oc rsh** command to access containers that are running on the OpenShift Container Platform cluster. Installing and running **sshd** in your image opens up additional vectors for attack and requirements for security patching.

Use volumes for persistent data

Images use a **volume** for persistent data. This way OpenShift Container Platform mounts the network storage to the node running the container, and if the container moves to a new node the storage is reattached to that node. By using the volume for all persistent storage needs, the content is preserved even if the container is restarted or moved. If your image writes data to arbitrary locations within the container, that content could not be preserved.

All data that needs to be preserved even after the container is destroyed must be written to a volume. Container engines support a **readonly** flag for containers, which can be used to strictly enforce good practices about not writing data to ephemeral storage in a container. Designing your image around that capability now makes it easier to take advantage of it later.

Explicitly defining volumes in your **Dockerfile** makes it easy for consumers of the image to understand what volumes they must define when running your image.

See the [Kubernetes documentation](#) for more information on how volumes are used in OpenShift Container Platform.



NOTE

Even with persistent volumes, each instance of your image has its own volume, and the filesystem is not shared between instances. This means the volume cannot be used to share state in a cluster.

4.1.2. OpenShift Container Platform-specific guidelines

The following are guidelines that apply when creating container images specifically for use on OpenShift Container Platform.

Enable images for source-to-image (S2I)

For images that are intended to run application code provided by a third party, such as a Ruby image designed to run Ruby code provided by a developer, you can enable your image to work with the [Source-to-Image \(S2I\)](#) build tool. S2I is a framework that makes it easy to write images that take application source code as an input and produce a new image that runs the assembled application as output.

Support arbitrary user ids

By default, OpenShift Container Platform runs containers using an arbitrarily assigned user ID. This provides additional security against processes escaping the container due to a container engine vulnerability and thereby achieving escalated permissions on the host node.

For an image to support running as an arbitrary user, directories and files that are written to by processes in the image must be owned by the root group and be read/writable by that group. Files to be executed must also have group execute permissions.

Adding the following to your Dockerfile sets the directory and file permissions to allow users in the root group to access them in the built image:

```
RUN chgrp -R 0 /some/directory && \  
    chmod -R g=u /some/directory
```

Because the container user is always a member of the root group, the container user can read and write these files.

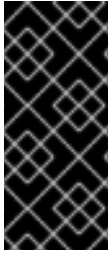


WARNING

Care must be taken when altering the directories and file permissions of sensitive areas of a container, which is no different than to a normal system.

If applied to sensitive areas, such as **/etc/passwd**, this can allow the modification of such files by unintended users potentially exposing the container or host. CRI-O supports the insertion of random user IDs into the container's **/etc/passwd**, so changing permissions is never required.

In addition, the processes running in the container must not listen on privileged ports, ports below 1024, since they are not running as a privileged user.



IMPORTANT

If your S2I image does not include a **USER** declaration with a numeric user, your builds fail by default. To allow images that use either named users or the root **0** user to build in OpenShift Container Platform, you can add the project's builder service account, **system:serviceaccount:<your-project>:builder**, to the **privileged** security context constraint (SCC). Alternatively, you can allow all images to run as any user.

Use services for inter-image communication

For cases where your image needs to communicate with a service provided by another image, such as a web front end image that needs to access a database image to store and retrieve data, your image consumes an OpenShift Container Platform service. Services provide a static endpoint for access which does not change as containers are stopped, started, or moved. In addition, services provide load balancing for requests.

Provide common libraries

For images that are intended to run application code provided by a third party, ensure that your image contains commonly used libraries for your platform. In particular, provide database drivers for common databases used with your platform. For example, provide JDBC drivers for MySQL and PostgreSQL if you are creating a Java framework image. Doing so prevents the need for common dependencies to be downloaded during application assembly time, speeding up application image builds. It also simplifies the work required by application developers to ensure all of their dependencies are met.

Use environment variables for configuration

Users of your image are able to configure it without having to create a downstream image based on your image. This means that the runtime configuration is handled using environment variables. For a simple configuration, the running process can consume the environment variables directly. For a more complicated configuration or for runtimes which do not support this, configure the runtime by defining a template configuration file that is processed during startup. During this processing, values supplied using environment variables can be substituted into the configuration file or used to make decisions about what options to set in the configuration file.

It is also possible and recommended to pass secrets such as certificates and keys into the container using environment variables. This ensures that the secret values do not end up committed in an image and leaked into a container image registry.

Providing environment variables allows consumers of your image to customize behavior, such as database settings, passwords, and performance tuning, without having to introduce a new layer on top of your image. Instead, they can simply define environment variable values when defining a pod and change those settings without rebuilding the image.

For extremely complex scenarios, configuration can also be supplied using volumes that would be mounted into the container at runtime. However, if you elect to do it this way you must ensure that your image provides clear error messages on startup when the necessary volume or configuration is not present.

This topic is related to the Using Services for Inter-image Communication topic in that configuration like datasources are defined in terms of environment variables that provide the service endpoint information. This allows an application to dynamically consume a datasource service that is defined in the OpenShift Container Platform environment without modifying the application image.

In addition, tuning is done by inspecting the **cgroups** settings for the container. This allows the image to tune itself to the available memory, CPU, and other resources. For example, Java-based images tune

their heap based on the **cgroup** maximum memory parameter to ensure they do not exceed the limits and get an out-of-memory error.

Set image metadata

Defining image metadata helps OpenShift Container Platform better consume your container images, allowing OpenShift Container Platform to create a better experience for developers using your image. For example, you can add metadata to provide helpful descriptions of your image, or offer suggestions on other images that are needed.

Clustering

You must fully understand what it means to run multiple instances of your image. In the simplest case, the load balancing function of a service handles routing traffic to all instances of your image. However, many frameworks must share information to perform leader election or failover state; for example, in session replication.

Consider how your instances accomplish this communication when running in OpenShift Container Platform. Although pods can communicate directly with each other, their IP addresses change anytime the pod starts, stops, or is moved. Therefore, it is important for your clustering scheme to be dynamic.

Logging

It is best to send all logging to standard out. OpenShift Container Platform collects standard out from containers and sends it to the centralized logging service where it can be viewed. If you must separate log content, prefix the output with an appropriate keyword, which makes it possible to filter the messages.

If your image logs to a file, users must use manual operations to enter the running container and retrieve or view the log file.

Liveness and readiness probes

Document example liveness and readiness probes that can be used with your image. These probes allow users to deploy your image with confidence that traffic is not be routed to the container until it is prepared to handle it, and that the container is restarted if the process gets into an unhealthy state.

Templates

Consider providing an example template with your image. A template gives users an easy way to quickly get your image deployed with a working configuration. Your template must include the liveness and readiness probes you documented with the image, for completeness.

4.2. INCLUDING METADATA IN IMAGES

Defining image metadata helps OpenShift Container Platform better consume your container images, allowing OpenShift Container Platform to create a better experience for developers using your image. For example, you can add metadata to provide helpful descriptions of your image, or offer suggestions on other images that may also be needed.

This topic only defines the metadata needed by the current set of use cases. Additional metadata or use cases may be added in the future.

4.2.1. Defining image metadata

You can use the **LABEL** instruction in a **Dockerfile** to define image metadata. Labels are similar to environment variables in that they are key value pairs attached to an image or a container. Labels are different from environment variable in that they are not visible to the running application and they can also be used for fast look-up of images and containers.

[Docker documentation](#) for more information on the **LABEL** instruction.

The label names are typically namespaced. The namespace is set accordingly to reflect the project that is going to pick up the labels and use them. For OpenShift Container Platform the namespace is set to **io.openshift** and for Kubernetes the namespace is **io.k8s**.

See the [Docker custom metadata](#) documentation for details about the format.

Table 4.1. Supported Metadata

Variable	Description
io.openshift.tags	<p>This label contains a list of tags represented as a list of comma-separated string values. The tags are the way to categorize the container images into broad areas of functionality. Tags help UI and generation tools to suggest relevant container images during the application creation process.</p> <pre> LABEL io.openshift.tags mongodb,mongodb24,nosql </pre>
io.openshift.wants	<p>Specifies a list of tags that the generation tools and the UI uses to provide relevant suggestions if you do not have the container images with specified tags already. For example, if the container image wants mysql and redis and you do not have the container image with redis tag, then UI can suggest you to add this image into your deployment.</p> <pre> LABEL io.openshift.wants mongodb,redis </pre>
io.k8s.description	<p>This label can be used to give the container image consumers more detailed information about the service or functionality this image provides. The UI can then use this description together with the container image name to provide more human friendly information to end users.</p> <pre> LABEL io.k8s.description The MySQL 5.5 Server with master-slave replication support </pre>
io.openshift.non-scalable	<p>An image can use this variable to suggest that it does not support scaling. The UI then communicates this to consumers of that image. Being not-scalable means that the value of replicas should initially not be set higher than 1.</p> <pre> LABEL io.openshift.non-scalable true </pre>
io.openshift.min-memory and io.openshift.min-cpu	<p>This label suggests how much resources the container image needs to work properly. The UI can warn the user that deploying this container image may exceed their user quota. The values must be compatible with Kubernetes quantity.</p> <pre> LABEL io.openshift.min-memory 8Gi LABEL io.openshift.min-cpu 4 </pre>

4.3. CREATING IMAGES FROM SOURCE CODE WITH SOURCE-TO-IMAGE

Source-to-image (S2I) is a framework that makes it easy to write images that take application source code as an input and produce a new image that runs the assembled application as output.

The main advantage of using S2I for building reproducible container images is the ease of use for developers. As a builder image author, you must understand two basic concepts in order for your images to provide the best S2I performance, the build process and S2I scripts.

4.3.1. Understanding the source-to-image build process

The build process consists of the following three fundamental elements, which are combined into a final container image:

- Sources
- Source-to-image (S2I) scripts
- Builder image

S2I generates a Dockerfile with the builder image as the first **FROM** instruction. The Dockerfile generated by S2I is then passed to Buildah.

4.3.2. How to write source-to-image scripts

You can write source-to-image (S2I) scripts in any programming language, as long as the scripts are executable inside the builder image. S2I supports multiple options providing **assemble/run/save-artifacts** scripts. All of these locations are checked on each build in the following order:


1. A script specified in the build configuration.
2. A script found in the application source **.s2i/bin** directory.
3. A script found at the default image URL with the **io.openshift.s2i.scripts-url** label.

Both the **io.openshift.s2i.scripts-url** label specified in the image and the script specified in a build configuration can take one of the following forms:

- **image:///path_to_scripts_dir**: absolute path inside the image to a directory where the S2I scripts are located.
- **file:///path_to_scripts_dir**: relative or absolute path to a directory on the host where the S2I scripts are located.
- **http(s)://path_to_scripts_dir**: URL to a directory where the S2I scripts are located.

Table 4.2. S2I scripts

Script	Description
--------	-------------

Script	Description
assemble	<p>The assemble script builds the application artifacts from a source and places them into appropriate directories inside the image. This script is required. The workflow for this script is:</p> <ol style="list-style-type: none"> 1. Optional: Restore build artifacts. If you want to support incremental builds, make sure to define save-artifacts as well. 2. Place the application source in the desired location. 3. Build the application artifacts. 4. Install the artifacts into locations appropriate for them to run.
run	The run script executes your application. This script is required.
save-artifacts	<p>The save-artifacts script gathers all dependencies that can speed up the build processes that follow. This script is optional. For example:</p> <ul style="list-style-type: none"> • For Ruby, gems installed by Bundler. • For Java, .m2 contents. <p>These dependencies are gathered into a tar file and streamed to the standard output.</p>
usage	The usage script allows you to inform the user how to properly use your image. This script is optional.
test/run	<p>The test/run script allows you to create a process to check if the image is working correctly. This script is optional. The proposed flow of that process is:</p> <ol style="list-style-type: none"> 1. Build the image. 2. Run the image to verify the usage script. 3. Run s2i build to verify the assemble script. 4. Optional: Run s2i build again to verify the save-artifacts and assemble scripts save and restore artifacts functionality. 5. Run the image to verify the test application is working. <div style="display: flex; align-items: flex-start; margin-top: 10px;"> <div style="flex: 1;">  </div> <div style="flex: 2;"> <p>NOTE</p> <p>The suggested location to put the test application built by your test/run script is the test/test-app directory in your image repository.</p> </div> </div>

Example S2I scripts

The following example S2I scripts are written in Bash. Each example assumes its **tar** contents are unpacked into the **/tmp/s2i** directory.

assemble script:

```
#!/bin/bash

# restore build artifacts
if [ "$(ls /tmp/s2i/artifacts/ 2>/dev/null)" ]; then
    mv /tmp/s2i/artifacts/* $HOME/.
fi

# move the application source
mv /tmp/s2i/src $HOME/src

# build application artifacts
pushd ${HOME}
make all

# install the artifacts
make install
popd
```

run script:

```
#!/bin/bash

# run the application
/opt/application/run.sh
```

save-artifacts script:

```
#!/bin/bash

pushd ${HOME}
if [ -d deps ]; then
    # all deps contents to tar stream
    tar cf - deps
fi
popd
```

usage script:

```
#!/bin/bash

# inform the user how to use the image
cat <<EOF
This is a S2I sample builder image, to use it, install
https://github.com/openshift/source-to-image
EOF
```

Additional resources

- [S2I Image Creation Tutorial](#)

4.4. ABOUT TESTING SOURCE-TO-IMAGE IMAGES

As an Source-to-Image (S2I) builder image author, you can test your S2I image locally and use the OpenShift Container Platform build system for automated testing and continuous integration.

S2I requires the **assemble** and **run** scripts to be present in order to successfully run the S2I build. Providing the **save-artifacts** script reuses the build artifacts, and providing the **usage** script ensures that usage information is printed to console when someone runs the container image outside of the S2I.

The goal of testing an S2I image is to make sure that all of these described commands work properly, even if the base container image has changed or the tooling used by the commands was updated.

4.4.1. Understanding testing requirements

The standard location for the **test** script is **test/run**. This script is invoked by the OpenShift Container Platform S2I image builder and it could be a simple Bash script or a static Go binary.

The **test/run** script performs the S2I build, so you must have the S2I binary available in your **\$PATH**. If required, follow the installation instructions in the [S2I README](#).

S2I combines the application source code and builder image, so in order to test it you need a sample application source to verify that the source successfully transforms into a runnable container image. The sample application should be simple, but it should exercise the crucial steps of **assemble** and **run** scripts.

4.4.2. Generating scripts and tools

The S2I tooling comes with powerful generation tools to speed up the process of creating a new S2I image. The **s2i create** command produces all the necessary S2I scripts and testing tools along with the **Makefile**:

```
$ s2i create _<image name>_ _<destination directory>_
```

The generated **test/run** script must be adjusted to be useful, but it provides a good starting point to begin developing.



NOTE

The **test/run** script produced by the **s2i create** command requires that the sample application sources are inside the **test/test-app** directory.

4.4.3. Testing locally

The easiest way to run the S2I image tests locally is to use the generated **Makefile**.

If you did not use the **s2i create** command, you can copy the following **Makefile** template and replace the **IMAGE_NAME** parameter with your image name.

Sample Makefile

```
IMAGE_NAME = openshift/ruby-20-centos7
CONTAINER_ENGINE := $(shell command -v podman 2> /dev/null | echo docker)

build:
```

```
{CONTAINER_ENGINE} build -t ${IMAGE_NAME} .
```

```
.PHONY: test
```

```
test:
```

```
{CONTAINER_ENGINE} build -t ${IMAGE_NAME}-candidate .
```

```
IMAGE_NAME=${IMAGE_NAME}-candidate test/run
```

4.4.4. Basic testing workflow

The **test** script assumes you have already built the image you want to test. If required, first build the S2I image. Run one of the following commands:

- If you use Podman, run the following command:

```
$ podman build -t <builder_image_name>
```

- If you use Docker, run the following command:

```
$ docker build -t <builder_image_name>
```

The following steps describe the default workflow to test S2I image builders:

1. Verify the **usage** script is working:

- If you use Podman, run the following command:

```
$ podman run <builder_image_name> .
```

- If you use Docker, run the following command:

```
$ docker run <builder_image_name> .
```

2. Build the image:

```
$ s2i build file:///path-to-sample-app _<BUILDER_IMAGE_NAME>_
_<OUTPUT_APPLICATION_IMAGE_NAME>_
```

3. Optional: if you support **save-artifacts**, run step 2 once again to verify that saving and restoring artifacts works properly.

4. Run the container:

- If you use Podman, run the following command:

```
$ podman run <output_application_image_name>
```

- If you use Docker, run the following command:

```
$ docker run <output_application_image_name>
```

5. Verify the container is running and the application is responding.

Running these steps is generally enough to tell if the builder image is working as expected.

4.4.5. Using OpenShift Container Platform for building the image

Once you have a **Dockerfile** and the other artifacts that make up your new S2I builder image, you can put them in a git repository and use OpenShift Container Platform to build and push the image. Define a Docker build that points to your repository.

If your OpenShift Container Platform instance is hosted on a public IP address, the build can be triggered each time you push into your S2I builder image GitHub repository.

You can also use the **ImageChangeTrigger** to trigger a rebuild of your applications that are based on the S2I builder image you updated.

CHAPTER 5. MANAGING IMAGES

5.1. MANAGING IMAGES OVERVIEW

With OpenShift Container Platform you can interact with images and set up image streams, depending on where the registries of the images are located, any authentication requirements around those registries, and how you want your builds and deployments to behave.

5.1.1. Images overview

An image stream comprises any number of container images identified by tags. It presents a single virtual view of related images, similar to a container image repository.

By watching an image stream, builds and deployments can receive notifications when new images are added or modified and react by performing a build or deployment, respectively.

5.2. TAGGING IMAGES

The following sections provide an overview and instructions for using image tags in the context of container images for working with OpenShift Container Platform image streams and their tags.

5.2.1. Image tags

An image tag is a label applied to a container image in a repository that distinguishes a specific image from other images in an image stream. Typically, the tag represents a version number of some sort. For example, here **:v3.11.59-2** is the tag:

```
registry.access.redhat.com/openshift3/jenkins-2-rhel7:v3.11.59-2
```

You can add additional tags to an image. For example, an image might be assigned the tags **:v3.11.59-2** and **:latest**.

OpenShift Container Platform provides the **oc tag** command, which is similar to the **docker tag** command, but operates on image streams instead of directly on images.

5.2.2. Image tag conventions

Images evolve over time and their tags reflect this. Generally, an image tag always points to the latest image built.

If there is too much information embedded in a tag name, like **v2.0.1-may-2019**, the tag points to just one revision of an image and is never updated. Using default image pruning options, such an image is never removed. In very large clusters, the schema of creating new tags for every revised image could eventually fill up the etcd datastore with excess tag metadata for images that are long outdated.

If the tag is named **v2.0**, image revisions are more likely. This results in longer tag history and, therefore, the image pruner is more likely to remove old and unused images.

Although tag naming convention is up to you, here are a few examples in the format **<image_name>:<image_tag>**:

Table 5.1. Image tag naming conventions

Description	Example
Revision	myimage:v2.0.1
Architecture	myimage:v2.0-x86_64
Base image	myimage:v1.2-centos7
Latest (potentially unstable)	myimage:latest
Latest stable	myimage:stable

If you require dates in tag names, periodically inspect old and unsupported images and **istags** and remove them. Otherwise, you can experience increasing resource usage caused by retaining old images.

5.2.3. Adding tags to image streams

An image stream in OpenShift Container Platform comprises zero or more container images identified by tags.

There are different types of tags available. The default behavior uses a **permanent** tag, which points to a specific image in time. If the **permanent** tag is in use and the source changes, the tag does not change for the destination.

A **tracking** tag means the destination tag's metadata is updated during the import of the source tag.

Procedure

- You can add tags to an image stream using the **oc tag** command:

```
$ oc tag <source> <destination>
```

For example, to configure the **ruby** image stream **static-2.0** tag to always refer to the current image for the **ruby** image stream **2.0** tag:

```
$ oc tag ruby:2.0 ruby:static-2.0
```

This creates a new image stream tag named **static-2.0** in the **ruby** image stream. The new tag directly references the image id that the **ruby:2.0** image stream tag pointed to at the time **oc tag** was run, and the image it points to never changes.

- To ensure the destination tag is updated when the source tag changes, use the **--alias=true** flag:

```
$ oc tag --alias=true <source> <destination>
```



NOTE

Use a tracking tag for creating permanent aliases, for example, **latest** or **stable**. The tag only works correctly within a single image stream. Trying to create a cross-image stream alias produces an error.

- You can also add the **--scheduled=true** flag to have the destination tag be refreshed, or re-imported, periodically. The period is configured globally at the system level.
- The **--reference** flag creates an image stream tag that is not imported. The tag points to the source location, permanently.
If you want to instruct OpenShift Container Platform to always fetch the tagged image from the integrated registry, use **--reference-policy=local**. The registry uses the pull-through feature to serve the image to the client. By default, the image blobs are mirrored locally by the registry. As a result, they can be pulled more quickly the next time they are needed. The flag also allows for pulling from insecure registries without a need to supply **--insecure-registry** to the container runtime as long as the image stream has an insecure annotation or the tag has an insecure import policy.

5.2.4. Removing tags from image streams

You can remove tags from an image stream.

Procedure

- To remove a tag completely from an image stream run:

```
$ oc delete istag/ruby:latest
```

or:

```
$ oc tag -d ruby:latest
```

5.2.5. Referencing images in imagestreams

You can use tags to reference images in image streams using the following reference types.

Table 5.2. Imagestream reference types

Reference type	Description
ImageStreamTag	An ImageStreamTag is used to reference or retrieve an image for a given image stream and tag.
ImageStreamImage	An ImageStreamImage is used to reference or retrieve an image for a given image stream and image sha ID.
DockerImage	A DockerImage is used to reference or retrieve an image for a given external registry. It uses standard Docker pull specification for its name.

When viewing example image stream definitions you may notice they contain definitions of **ImageStreamTag** and references to **DockerImage**, but nothing related to **ImageStreamImage**.

This is because the **ImageStreamImage** objects are automatically created in OpenShift Container Platform when you import or tag an image into the image stream. You should never have to explicitly define an **ImageStreamImage** object in any image stream definition that you use to create image

streams.

Procedure

- To reference an image for a given image stream and tag, use **ImageStreamTag**:

```
<image_stream_name>:<tag>
```

- To reference an image for a given image stream and image **sha** ID, use **ImageStreamImage**:

```
<image_stream_name>@<id>
```

The **<id>** is an immutable identifier for a specific image, also called a digest.

- To reference or retrieve an image for a given external registry, use **DockerImage**:

```
openshift/ruby-20-centos7:2.0
```



NOTE

When no tag is specified, it is assumed the **latest** tag is used.

You can also reference a third-party registry:

```
registry.redhat.io/rhel7:latest
```

Or an image with a digest:

```
centos/ruby-22-  
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b2  
8e
```

5.3. IMAGE PULL POLICY

Each container in a pod has a container image. Once you have created an image and pushed it to a registry, you can then refer to it in the pod.

5.3.1. Image pull policy overview

When OpenShift Container Platform creates containers, it uses the container **imagePullPolicy** to determine if the image should be pulled prior to starting the container. There are three possible values for **imagePullPolicy**:

Table 5.3. **imagePullPolicy** values

Value	Description
Always	Always pull the image.
IfNotPresent	Only pull the image if it does not already exist on the node.

Value	Description
Never	Never pull the image.

If a container **imagePullPolicy** parameter is not specified, OpenShift Container Platform sets it based on the image tag:

1. If the tag is **latest**, OpenShift Container Platform defaults **imagePullPolicy** to **Always**.
2. Otherwise, OpenShift Container Platform defaults **imagePullPolicy** to **IfNotPresent**.

5.4. USING IMAGE PULL SECRETS

If you are using the OpenShift Container Platform internal registry and are pulling from image streams located in the same project, then your pod service account should already have the correct permissions and no additional action should be required.

However, for other scenarios, such as referencing images across OpenShift Container Platform projects or from secured registries, then additional configuration steps are required.

You can obtain the image pull secret, **pullSecret**, from the [Pull Secret](#) page on the Red Hat OpenShift Cluster Manager site.

You use this pull secret to authenticate with the services that are provided by the included authorities, including [Quay.io](#) and [registry.redhat.io](#), which serve the container images for OpenShift Container Platform components.

Example config.json file

```
{
  "auths":{
    "cloud.openshift.com":{
      "auth":"b3Blb=",
      "email":"you@example.com"
    },
    "quay.io":{
      "auth":"b3Blb=",
      "email":"you@example.com"
    }
  }
}
```

5.4.1. Allowing pods to reference images across projects

When using the internal registry, to allow pods in **project-a** to reference images in **project-b**, a service account in **project-a** must be bound to the **system:image-puller** role in **project-b**.

Procedure

1. To allow pods in **project-a** to reference images in **project-b**, bind a service account in **project-a** to the **system:image-puller** role in **project-b**:

```
$ oc policy add-role-to-user \
  system:image-puller system:serviceaccount:project-a:default \
  --namespace=project-b
```

After adding that role, the pods in **project-a** that reference the default service account are able to pull images from **project-b**.

2. To allow access for any service account in **project-a**, use the group:

```
$ oc policy add-role-to-group \
  system:image-puller system:serviceaccounts:project-a \
  --namespace=project-b
```

5.4.2. Allowing pods to reference images from other secured registries

The **.dockercfg** **\$HOME/.docker/config.json** file for Docker clients is a Docker credentials file that stores your authentication information if you have previously logged into a secured or insecure registry.

To pull a secured container image that is not from OpenShift Container Platform's internal registry, you must create a pull secret from your Docker credentials and add it to your service account.

Procedure

- If you already have a **.dockercfg** file for the secured registry, you can create a secret from that file by running:

```
$ oc create secret generic <pull_secret_name> \
  --from-file=.dockercfg=<path/to/.dockercfg> \
  --type=kubernetes.io/dockercfg
```

- Or if you have a **\$HOME/.docker/config.json** file:

```
$ oc create secret generic <pull_secret_name> \
  --from-file=.dockerconfigjson=<path/to/.docker/config.json> \
  --type=kubernetes.io/dockerconfigjson
```

- If you do not already have a Docker credentials file for the secured registry, you can create a secret by running:

```
$ oc create secret docker-registry <pull_secret_name> \
  --docker-server=<registry_server> \
  --docker-username=<user_name> \
  --docker-password=<password> \
  --docker-email=<email>
```

- To use a secret for pulling images for pods, you must add the secret to your service account. The name of the service account in this example should match the name of the service account the pod uses. The default service account is **default**:

```
$ oc secrets link default <pull_secret_name> --for=pull
```

5.4.2.1. Pulling from private registries with delegated authentication

A private registry can delegate authentication to a separate service. In these cases, image pull secrets must be defined for both the authentication and registry endpoints.

Procedure

1. Create a secret for the delegated authentication server:

```
$ oc create secret docker-registry \  
  --docker-server=sso.redhat.com \  
  --docker-username=developer@example.com \  
  --docker-password=***** \  
  --docker-email=unused \  
  redhat-connect-sso  
  
secret/redhat-connect-sso
```

2. Create a secret for the private registry:

```
$ oc create secret docker-registry \  
  --docker-server=privateregistry.example.com \  
  --docker-username=developer@example.com \  
  --docker-password=***** \  
  --docker-email=unused \  
  private-registry  
  
secret/private-registry
```

5.4.3. Updating the global cluster pull secret

You can update the global pull secret for your cluster.



WARNING

Cluster resources must adjust to the new pull secret, which can temporarily limit the usability of the cluster.



WARNING

Updating the global pull secret will cause node reboots while the Machine Config Operator (MCO) syncs the changes.

Prerequisites

- You have a new or modified pull secret file to upload.
- You have access to the cluster as a user with the **cluster-admin** role.

Procedure

- Enter the following command to update the global pull secret for your cluster:

```
$ oc set data secret/pull-secret -n openshift-config --from-file=.dockerconfigjson=<pull-secret-location> 1
```

- 1** Provide the path to the new pull secret file.

This update is rolled out to all nodes, which can take some time depending on the size of your cluster. During this time, nodes are drained and pods are rescheduled on the remaining nodes.

CHAPTER 6. MANAGING IMAGE STREAMS

Image streams provide a means of creating and updating container images in an on-going way. As improvements are made to an image, tags can be used to assign new version numbers and keep track of changes. This document describes how image streams are managed.

6.1. USING IMAGE STREAMS

An image stream and its associated tags provide an abstraction for referencing container images from within OpenShift Container Platform. The image stream and its tags allow you to see what images are available and ensure that you are using the specific image you need even if the image in the repository changes.

Image streams do not contain actual image data, but present a single virtual view of related images, similar to an image repository.

You can configure builds and deployments to watch an image stream for notifications when new images are added and react by performing a build or deployment, respectively.

For example, if a deployment is using a certain image and a new version of that image is created, a deployment could be automatically performed to pick up the new version of the image.

However, if the image stream tag used by the deployment or build is not updated, then even if the container image in the container image registry is updated, the build or deployment continues using the previous, presumably known good image.

The source images can be stored in any of the following:

- OpenShift Container Platform's integrated registry.
- An external registry, for example registry.redhat.io or [Quay.io](https://quay.io).
- Other image streams in the OpenShift Container Platform cluster.

When you define an object that references an image stream tag, such as a build or deployment configuration, you point to an image stream tag and not the repository. When you build or deploy your application, OpenShift Container Platform queries the repository using the image stream tag to locate the associated ID of the image and uses that exact image.

The image stream metadata is stored in the etcd instance along with other cluster information.

Using image streams has several significant benefits:

- You can tag, rollback a tag, and quickly deal with images, without having to re-push using the command line.
- You can trigger builds and deployments when a new image is pushed to the registry. Also, OpenShift Container Platform has generic triggers for other resources, such as Kubernetes objects.
- You can mark a tag for periodic re-import. If the source image has changed, that change is picked up and reflected in the image stream, which triggers the build or deployment flow, depending upon the build or deployment configuration.
- You can share images using fine-grained access control and quickly distribute images across your teams.

- If the source image changes, the image stream tag still points to a known-good version of the image, ensuring that your application do not break unexpectedly.
- You can configure security around who can view and use the images through permissions on the image stream objects.
- Users that lack permission to read or list images on the cluster level can still retrieve the images tagged in a project using image streams.

6.2. CONFIGURING IMAGE STREAMS

An **ImageStream** object file contains the following elements.

Imagestream object definition

```

apiVersion: v1
kind: ImageStream
metadata:
  annotations:
    openshift.io/generated-by: OpenShiftNewApp
  creationTimestamp: 2017-09-29T13:33:49Z
  generation: 1
  labels:
    app: ruby-sample-build
    template: application-template-stibuild
  name: origin-ruby-sample ❶
  namespace: test
  resourceVersion: "633"
  selflink: /oapi/v1/namespaces/test/imagestreams/origin-ruby-sample
  uid: ee2b9405-c68c-11e5-8a99-525400f25e34
spec: {}
status:
  dockerImageRepository: 172.30.56.218:5000/test/origin-ruby-sample ❷
  tags:
    - items:
      - created: 2017-09-02T10:15:09Z
        dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d ❸
        generation: 2
        image: sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab3fc5 ❹
      - created: 2017-09-29T13:40:11Z
        dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab3fc5
        generation: 1
        image: sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d
        tag: latest ❺

```

❶ The name of the image stream.

❷ Docker repository path where new images can be pushed to add or update them in this image stream.

❸ The SHA identifier that this image stream tag currently references. Resources that reference this image stream tag use this identifier.

- 4 The SHA identifier that this image stream tag previously referenced. Can be used to rollback to an older image.
- 5 The image stream tag name.

6.3. IMAGE STREAM IMAGES

An image stream image points from within an image stream to a particular image ID.

Image stream images allow you to retrieve metadata about an image from a particular image stream where it is tagged.

Image stream image objects are automatically created in OpenShift Container Platform whenever you import or tag an image into the image stream. You should never have to explicitly define an image stream image object in any image stream definition that you use to create image streams.

The image stream image consists of the image stream name and image ID from the repository, delimited by an @ sign:

```
<image-stream-name>@<image-id>
```

To refer to the image in the **ImageStream** object example, the image stream image looks like:

```
origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d
```

6.4. IMAGE STREAM TAGS

An image stream tag is a named pointer to an image in an image stream. It is abbreviated as **istag**. An image stream tag is used to reference or retrieve an image for a given image stream and tag.

Image stream tags can reference any local or externally managed image. It contains a history of images represented as a stack of all images the tag ever pointed to. Whenever a new or existing image is tagged under particular image stream tag, it is placed at the first position in the history stack. The image previously occupying the top position is available at the second position. This allows for easy rollbacks to make tags point to historical images again.

The following image stream tag is from an **ImageStream** object:

Image stream tag with two images in its history

```
tags:
- items:
  - created: 2017-09-02T10:15:09Z
    dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d
  generation: 2
  image: sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab3fc5
  - created: 2017-09-29T13:40:11Z
    dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab3fc5
```

```

generation: 1
image: sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d
tag: latest

```

Image stream tags can be permanent tags or tracking tags.

- Permanent tags are version-specific tags that point to a particular version of an image, such as Python 3.5.
- Tracking tags are reference tags that follow another image stream tag and can be updated to change which image they follow, like a symlink. These new levels are not guaranteed to be backwards-compatible.

For example, the **latest** image stream tags that ship with OpenShift Container Platform are tracking tags. This means consumers of the **latest** image stream tag are updated to the newest level of the framework provided by the image when a new level becomes available. A **latest** image stream tag to **v3.10** can be changed to **v3.11** at any time. It is important to be aware that these **latest** image stream tags behave differently than the Docker **latest** tag. The **latest** image stream tag, in this case, does not point to the latest image in the Docker repository. It points to another image stream tag, which might not be the latest version of an image. For example, if the **latest** image stream tag points to **v3.10** of an image, when the **3.11** version is released, the **latest** tag is not automatically updated to **v3.11**, and remains at **v3.10** until it is manually updated to point to a **v3.11** image stream tag.



NOTE

Tracking tags are limited to a single image stream and cannot reference other image streams.

You can create your own image stream tags for your own needs.

The image stream tag is composed of the name of the image stream and a tag, separated by a colon:

```
<imagestream name>:<tag>
```

For example, to refer to the **sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d** image in the **ImageStream** object example earlier, the image stream tag would be:

```
origin-ruby-sample:latest
```

6.5. IMAGE STREAM CHANGE TRIGGERS

Image stream triggers allow your builds and deployments to be automatically invoked when a new version of an upstream image is available.

For example, builds and deployments can be automatically started when an image stream tag is modified. This is achieved by monitoring that particular image stream tag and notifying the build or deployment when a change is detected.

6.6. IMAGE STREAM MAPPING

When the integrated registry receives a new image, it creates and sends an image stream mapping to OpenShift Container Platform, providing the image's project, name, tag, and image metadata.

**NOTE**

Configuring image stream mappings is an advanced feature.

This information is used to create a new image, if it does not already exist, and to tag the image into the image stream. OpenShift Container Platform stores complete metadata about each image, such as commands, entry point, and environment variables. Images in OpenShift Container Platform are immutable and the maximum name length is 63 characters.

The following image stream mapping example results in an image being tagged as **test/origin-ruby-sample:latest**:

Image stream mapping object definition

```

apiVersion: v1
kind: ImageStreamMapping
metadata:
  creationTimestamp: null
  name: origin-ruby-sample
  namespace: test
tag: latest
image:
  dockerImageLayers:
  - name: sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef
    size: 0
  - name: sha256:ee1dd2cb6df21971f4af6de0f1d7782b81fb63156801cfde2bb47b4247c23c29
    size: 196634330
  - name: sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef
    size: 0
  - name: sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef
    size: 0
  - name: sha256:ca062656bff07f18bff46be00f40cfbb069687ec124ac0aa038fd676cfaea092
    size: 177723024
  - name: sha256:63d529c59c92843c395befd065de516ee9ed4995549f8218eac6ff088bfa6b6e
    size: 55679776
  - name: sha256:92114219a04977b5563d7dff71ec4caa3a37a15b266ce42ee8f43dba9798c966
    size: 11939149
  dockerImageMetadata:
    Architecture: amd64
    Config:
      Cmd:
      - /usr/libexec/s2i/run
      Entrypoint:
      - container-entrypoint
      Env:
      - RACK_ENV=production
      - OPENSIFT_BUILD_NAMESPACE=test
      - OPENSIFT_BUILD_SOURCE=https://github.com/openshift/ruby-hello-world.git
      - EXAMPLE=sample-app
      - OPENSIFT_BUILD_NAME=ruby-sample-build-1
      - PATH=/opt/app-root/src/bin:/opt/app-
root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
      - STI_SCRIPTS_URL=image:///usr/libexec/s2i
      - STI_SCRIPTS_PATH=/usr/libexec/s2i
      - HOME=/opt/app-root/src
      - BASH_ENV=/opt/app-root/etc/scl_enable

```

```

- ENV=/opt/app-root/etc/scl_enable
- PROMPT_COMMAND=. /opt/app-root/etc/scl_enable
- RUBY_VERSION=2.2
ExposedPorts:
  8080/tcp: {}
Labels:
  build-date: 2015-12-23
  io.k8s.description: Platform for building and running Ruby 2.2 applications
  io.k8s.display-name: 172.30.56.218:5000/test/origin-ruby-sample:latest
  io.openshift.build.commit.author: Ben Parees <bparees@users.noreply.github.com>
  io.openshift.build.commit.date: Wed Jan 20 10:14:27 2016 -0500
  io.openshift.build.commit.id: 00cadc392d39d5ef9117cbc8a31db0889eedd442
  io.openshift.build.commit.message: 'Merge pull request #51 from php-coder/fix_url_and_sti'
  io.openshift.build.commit.ref: master
  io.openshift.build.image: centos/ruby-22-
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b28e
  io.openshift.build.source-location: https://github.com/openshift/ruby-hello-world.git
  io.openshift.builder-base-version: 8d95148
  io.openshift.builder-version: 8847438ba06307f86ac877465eadc835201241df
  io.openshift.s2i.scripts-url: image:///usr/libexec/s2i
  io.openshift.tags: builder,ruby,ruby22
  io.s2i.scripts-url: image:///usr/libexec/s2i
  license: GPLv2
  name: CentOS Base Image
  vendor: CentOS
User: "1001"
WorkingDir: /opt/app-root/src
Container: 86e9a4a3c760271671ab913616c51c9f3cea846ca524bf07c04a6f6c9e103a76
ContainerConfig:
  AttachStdout: true
  Cmd:
  - /bin/sh
  - -c
  - tar -C /tmp -xf - && /usr/libexec/s2i/assemble
  Entrypoint:
  - container-entrypoint
  Env:
  - RACK_ENV=production
  - OPENSIFT_BUILD_NAME=ruby-sample-build-1
  - OPENSIFT_BUILD_NAMESPACE=test
  - OPENSIFT_BUILD_SOURCE=https://github.com/openshift/ruby-hello-world.git
  - EXAMPLE=sample-app
  - PATH=/opt/app-root/src/bin:/opt/app-
root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
  - STI_SCRIPTS_URL=image:///usr/libexec/s2i
  - STI_SCRIPTS_PATH=/usr/libexec/s2i
  - HOME=/opt/app-root/src
  - BASH_ENV=/opt/app-root/etc/scl_enable
  - ENV=/opt/app-root/etc/scl_enable
  - PROMPT_COMMAND=. /opt/app-root/etc/scl_enable
  - RUBY_VERSION=2.2
  ExposedPorts:
  8080/tcp: {}
  Hostname: ruby-sample-build-1-build
  Image: centos/ruby-22-
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b28e

```

```

OpenStdin: true
StdinOnce: true
User: "1001"
WorkingDir: /opt/app-root/src
Created: 2016-01-29T13:40:00Z
DockerVersion: 1.8.2.fc21
Id: 9d7fd5e2d15495802028c569d544329f4286dcd1c9c085ff5699218dbaa69b43
Parent: 57b08d979c86f4500dc8cad639c9518744c8dd39447c055a3517dc9c18d6fccd
Size: 441976279
apiVersion: "1.0"
kind: DockerImage
dockerImageMetadataVersion: "1.0"
dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d

```

6.7. WORKING WITH IMAGE STREAMS

The following sections describe how to use image streams and image stream tags.

6.7.1. Getting information about image streams

You can get general information about the image stream and detailed information about all the tags it is pointing to.

Procedure

- Get general information about the image stream and detailed information about all the tags it is pointing to:

```
$ oc describe is/<image-name>
```

For example:

```
$ oc describe is/python
```

Example output

```

Name: python
Namespace: default
Created: About a minute ago
Labels: <none>
Annotations: openshift.io/image.dockerRepositoryCheck=2017-10-02T17:05:11Z
Docker Pull Spec: docker-registry.default.svc:5000/default/python
Image Lookup: local=false
Unique Images: 1
Tags: 1

3.5
  tagged from centos/python-35-centos7

* centos/python-35-
centos7@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
  About a minute ago

```


- Get all the information available about particular image stream tag:

```
$ oc describe istag/<image-stream>:<tag-name>
```

For example:

```
$ oc describe istag/python:latest
```

Example output

```
Image Name: sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
Docker Image: centos/python-35-
centos7@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
Name: sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
Created: 2 minutes ago
Image Size: 251.2 MB (first layer 2.898 MB, last binary layer 72.26 MB)
Image Created: 2 weeks ago
Author: <none>
Arch: amd64
Entrypoint: container-entrypoint
Command: /bin/sh -c $STI_SCRIPTS_PATH/usage
Working Dir: /opt/app-root/src
User: 1001
Exposes Ports: 8080/tcp
Docker Labels: build-date=20170801
```



NOTE

More information is output than shown.

6.7.2. Adding tags to an image stream

You can add additional tags to image streams.

Procedure

- Add a tag that points to one of the existing tags by using the ``oc tag`` command:

```
$ oc tag <image-name:tag1> <image-name:tag2>
```

For example:

```
$ oc tag python:3.5 python:latest
```

Example output

```
Tag python:latest set to
python@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25.
```

- Confirm the image stream has two tags, one, **3.5**, pointing at the external container image and another tag, **latest**, pointing to the same image because it was created based on the first tag.

```
$ oc describe is/python
```

Example output

```
Name: python
Namespace: default
Created: 5 minutes ago
Labels: <none>
Annotations: openshift.io/image.dockerRepositoryCheck=2017-10-02T17:05:11Z
             Docker Pull Spec: docker-registry.default.svc:5000/default/python
             Image Lookup: local=false
             Unique Images: 1
             Tags: 2

latest
  tagged from
  python@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25

  * centos/python-35-
  centos7@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
    About a minute ago

3.5
  tagged from centos/python-35-centos7

  * centos/python-35-
  centos7@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
    5 minutes ago
```

6.7.3. Adding tags for an external image

You can add tags for external images.

Procedure

- Add tags pointing to internal or external images, by using the **oc tag** command for all tag-related operations:

```
$ oc tag <repository/image> <image-name:tag>
```

For example, this command maps the **docker.io/python:3.6.0** image to the **3.6** tag in the **python** image stream.

```
$ oc tag docker.io/python:3.6.0 python:3.6
```

Example output

```
Tag python:3.6 set to docker.io/python:3.6.0.
```

If the external image is secured, you must create a secret with credentials for accessing that registry.

6.7.4. Updating image stream tags

You can update a tag to reflect another tag in an image stream.

Procedure

- Update a tag:

```
$ oc tag <image-name:tag> <image-name:latest>
```

For example, the following updates the **latest** tag to reflect the **3.6** tag in an image stream:

```
$ oc tag python:3.6 python:latest
```

Example output

```
Tag python:latest set to
python@sha256:438208801c4806548460b27bd1fbc7bb188273d13871ab43f.
```

6.7.5. Removing image stream tags

You can remove old tags from an image stream.

Procedure

- Remove old tags from an image stream:

```
$ oc tag -d <image-name:tag>
```

For example:

```
$ oc tag -d python:3.5
```

Example output

```
Deleted tag default/python:3.5.
```

6.7.6. Configuring periodic importing of image stream tags

When working with an external container image registry, to periodically re-import an image, for example to get latest security updates, you can use the **--scheduled** flag.

Procedure

1. Schedule importing images:

```
$ oc tag <repository/image> <image-name:tag> --scheduled
```

For example:

```
$ oc tag docker.io/python:3.6.0 python:3.6 --scheduled
```

Example output

```
Tag python:3.6 set to import docker.io/python:3.6.0 periodically.
```

This command causes OpenShift Container Platform to periodically update this particular image stream tag. This period is a cluster-wide setting set to 15 minutes by default.

2. Remove the periodic check, re-run above command but omit the **--scheduled** flag. This release_notes its behavior to default.

```
$ oc tag <repository/image> <image-name:tag>
```

6.8. IMPORTING IMAGES AND IMAGE STREAMS FROM PRIVATE REGISTRIES

An image stream can be configured to import tag and image metadata from private image registries requiring authentication. This procedure applies if you change the registry that the Cluster Samples Operator uses to pull content from to something other than registry.redhat.io.



NOTE

When importing from insecure or secure registries, the registry URL defined in the secret must include the **:80** port suffix or the secret is not used when attempting to import from the registry.

Procedure

1. You must create a **secret** object that is used to store your credentials by entering the following command:

```
$ oc create secret generic <secret_name> --from-file=.dockerconfigjson=  
<file_absolute_path> --type=kubernetes.io/dockerconfigjson
```

2. After the secret is configured, create the new image stream or enter the **oc import-image** command:

```
$ oc import-image <imagestreamtag> --from=<image> --confirm
```

During the import process, OpenShift Container Platform picks up the secrets and provides them to the remote party.

6.8.1. Allowing pods to reference images from other secured registries

The **.dockercfg \$HOME/.docker/config.json** file for Docker clients is a Docker credentials file that stores your authentication information if you have previously logged into a secured or insecure registry.

To pull a secured container image that is not from OpenShift Container Platform's internal registry, you must create a pull secret from your Docker credentials and add it to your service account.

Procedure

- If you already have a **.dockercfg** file for the secured registry, you can create a secret from that file by running:

```
$ oc create secret generic <pull_secret_name> \  
  --from-file=.dockercfg=<path/to/.dockercfg> \  
  --type=kubernetes.io/dockercfg
```

- Or if you have a **\$HOME/.docker/config.json** file:

```
$ oc create secret generic <pull_secret_name> \  
  --from-file=.dockerconfigjson=<path/to/.docker/config.json> \  
  --type=kubernetes.io/dockerconfigjson
```

- If you do not already have a Docker credentials file for the secured registry, you can create a secret by running:

```
$ oc create secret docker-registry <pull_secret_name> \  
  --docker-server=<registry_server> \  
  --docker-username=<user_name> \  
  --docker-password=<password> \  
  --docker-email=<email>
```

- To use a secret for pulling images for pods, you must add the secret to your service account. The name of the service account in this example should match the name of the service account the pod uses. The default service account is **default**:

```
$ oc secrets link default <pull_secret_name> --for=pull
```

CHAPTER 7. USING IMAGE STREAMS WITH KUBERNETES RESOURCES

Image streams, being OpenShift Container Platform native resources, work out of the box with all the rest of native resources available in OpenShift Container Platform, such as builds or deployments. It is also possible to make them work with native Kubernetes resources, such as jobs, replication controllers, replica sets or Kubernetes deployments.

7.1. ENABLING IMAGE STREAMS WITH KUBERNETES RESOURCES

When using image streams with Kubernetes resources, you can only reference image streams that reside in the same project as the resource. The image stream reference must consist of a single segment value, for example **ruby:2.5**, where **ruby** is the name of an image stream that has a tag named **2.5** and resides in the same project as the resource making the reference.



NOTE

This feature can not be used in the **default** namespace, nor in any **openshift-** or **kube-** namespace.

There are two ways to enable image streams with Kubernetes resources:

- Enabling image stream resolution on a specific resource. This allows only this resource to use the image stream name in the image field.
- Enabling image stream resolution on an image stream. This allows all resources pointing to this image stream to use it in the image field.

Procedure

You can use **oc set image-lookup** to enable image stream resolution on a specific resource or image stream resolution on an image stream.

1. To allow all resources to reference the image stream named **mysql**, enter the following command:

```
$ oc set image-lookup mysql
```

This sets the **Imagestream.spec.lookupPolicy.local** field to true.

Imagestream with image lookup enabled

```
apiVersion: v1
kind: ImageStream
metadata:
  annotations:
    openshift.io/display-name: mysql
  name: mysql
  namespace: myproject
spec:
  lookupPolicy:
    local: true
```

When enabled, the behavior is enabled for all tags within the image stream.

2. Then you can query the image streams and see if the option is set:

```
$ oc set image-lookup imagestream --list
```

You can enable image lookup on a specific resource.

- To allow the Kubernetes deployment named **mysql** to use image streams, run the following command:

```
$ oc set image-lookup deploy/mysql
```

This sets the **alpha.image.policy.openshift.io/resolve-names** annotation on the deployment.

Deployment with image lookup enabled

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  namespace: myproject
spec:
  replicas: 1
  template:
    metadata:
      annotations:
        alpha.image.policy.openshift.io/resolve-names: '*'
    spec:
      containers:
      - image: mysql:latest
        imagePullPolicy: Always
        name: mysql
```

You can disable image lookup.

- To disable image lookup, pass **--enabled=false**:

```
$ oc set image-lookup deploy/mysql --enabled=false
```

CHAPTER 8. TRIGGERING UPDATES ON IMAGE STREAM CHANGES

When an image stream tag is updated to point to a new image, OpenShift Container Platform can automatically take action to roll the new image out to resources that were using the old image. You configure this behavior in different ways depending on the type of resource that references the image stream tag.

8.1. OPENSIFT CONTAINER PLATFORM RESOURCES

OpenShift Container Platform deployment configurations and build configurations can be automatically triggered by changes to image stream tags. The triggered action can be run using the new value of the image referenced by the updated image stream tag.

8.2. TRIGGERING KUBERNETES RESOURCES

Kubernetes resources do not have fields for triggering, unlike deployment and build configurations, which include as part of their API definition a set of fields for controlling triggers. Instead, you can use annotations in OpenShift Container Platform to request triggering.

The annotation is defined as follows:

```
Key: image.openshift.io/triggers
Value:
[
  {
    "from": {
      "kind": "ImageStreamTag", 1
      "name": "example:latest", 2
      "namespace": "myapp" 3
    },
    "fieldPath": "spec.template.spec.containers[?(@.name=='web')].image", 4
    "paused": "false" 5
  },
  ...
]
```

- 1 Required: **kind** is the resource to trigger from must be **ImageStreamTag**.
- 2 Required: **name** must be the name of an image stream tag.
- 3 Optional: **namespace** defaults to the namespace of the object.
- 4 Required: **fieldPath** is the JSON path to change. This field is limited and accepts only a JSON path expression that precisely matches a container by ID or index. For pods, the JSON path is "spec.containers[?(@.name='web')].image".
- 5 Optional: **paused** is whether or not the trigger is paused, and the default value is **false**. Set **paused** to **true** to temporarily disable this trigger.

When one of the core Kubernetes resources contains both a pod template and this annotation, OpenShift Container Platform attempts to update the object by using the image currently associated with the image stream tag that is referenced by trigger. The update is performed against the **fieldPath**

specified.

Examples of core Kubernetes resources that can contain both a pod template and annotation include:

- **CronJobs**
- **Deployments**
- **StatefulSets**
- **DaemonSets**
- **Jobs**
- **ReplicationControllers**
- **Pods**

8.3. SETTING THE IMAGE TRIGGER ON KUBERNETES RESOURCES

When adding an image trigger to deployments, you can use the **oc set triggers** command. For example, the sample command in this procedure adds an image change trigger to the deployment named **example** so that when the **example:latest** image stream tag is updated, the **web** container inside the deployment updates with the new image value. This command sets the correct **image.openshift.io/triggers** annotation on the deployment resource.

Procedure

- Trigger Kubernetes resources by entering the **oc set triggers** command:

```
$ oc set triggers deploy/example --from-image=example:latest -c web
```

Unless the deployment is paused, this pod template update automatically causes a deployment to occur with the new image value.

CHAPTER 9. IMAGE CONFIGURATION RESOURCES

Use the following procedure to configure image registries.

9.1. IMAGE CONTROLLER CONFIGURATION PARAMETERS

The **image.config.openshift.io/cluster** resource holds cluster-wide information about how to handle images. The canonical, and only valid name is **cluster**. Its **spec** offers the following configuration parameters.

Parameter	Description
allowedRegistriesForImport	<p>Limits the container image registries from which normal users can import images. Set this list to the registries that you trust to contain valid images, and that you want applications to be able to import from. Users with permission to create images or ImageStreamMappings from the API are not affected by this policy. Typically only cluster administrators have the appropriate permissions.</p> <p>Every element of this list contains a location of the registry specified by the registry domain name.</p> <p>domainName: Specifies a domain name for the registry. In case the registry uses a non-standard 80 or 443 port, the port should be included in the domain name as well.</p> <p>insecure: Insecure indicates whether the registry is secure or insecure. By default, if not otherwise specified, the registry is assumed to be secure.</p>
additionalTrustedCA	<p>A reference to a config map containing additional CAs that should be trusted during image stream import, pod image pull, openshift-image-registry pullthrough, and builds.</p> <p>The namespace for this config map is openshift-config. The format of the config map is to use the registry hostname as the key, and the PEM-encoded certificate as the value, for each additional registry CA to trust.</p>
externalRegistryHostnames	<p>Provides the host names for the default external image registry. The external hostname should be set only when the image registry is exposed externally. The first value is used in publicDockerImageRepository field in image streams. The value must be in hostname[:port] format.</p>

Parameter	Description
registrySources	<p>Contains configuration that determines how the container runtime should treat individual registries when accessing images for builds and pods. For instance, whether or not to allow insecure access. It does not contain configuration for the internal cluster registry.</p> <p>insecureRegistries: Registries which do not have a valid TLS certificate or only support HTTP connections.</p> <p>blockedRegistries: Denylist for image pull and push actions. All other registries are allowed.</p> <p>allowedRegistries: Allowlist for image pull and push actions. All other registries are blocked.</p> <p>Either blockedRegistries or allowedRegistries can be set, but not both.</p>



WARNING

When the **allowedRegistries** parameter is defined, all registries, including **registry.redhat.io** and **quay.io** registries and the default internal image registry, are blocked unless explicitly listed. When using the parameter, to prevent pod failure, add all registries including the **registry.redhat.io** and **quay.io** registries and the **internalRegistryHostname** to the **allowedRegistries** list, as they are required by payload images within your environment. For disconnected clusters, mirror registries should also be added.

The **status** field of the **image.config.openshift.io/cluster** resource holds observed values from the cluster.

Parameter	Description
internalRegistryHostname	Set by the Image Registry Operator, which controls the internalRegistryHostname . It sets the hostname for the default internal image registry. The value must be in hostname[:port] format. For backward compatibility, you can still use the OPENSIFT_DEFAULT_REGISTRY environment variable, but this setting overrides the environment variable.
externalRegistryHostnames	Set by the Image Registry Operator, provides the external host names for the image registry when it is exposed externally. The first value is used in publicDockerImageRepository field in image streams. The values must be in hostname[:port] format.

9.2. CONFIGURING IMAGE SETTINGS

You can configure image registry settings by editing the **image.config.openshift.io/cluster** custom resource (CR). The Machine Config Operator (MCO) watches the **image.config.openshift.io/cluster** CR for any changes to the registries and reboots the nodes when it detects changes.

Procedure

1. Edit the **image.config.openshift.io/cluster** custom resource:

```
$ oc edit image.config.openshift.io/cluster
```

The following is an example **image.config.openshift.io/cluster** CR:

```
apiVersion: config.openshift.io/v1
kind: Image 1
metadata:
  annotations:
    release.openshift.io/create-only: "true"
  creationTimestamp: "2019-05-17T13:44:26Z"
  generation: 1
  name: cluster
  resourceVersion: "8302"
  selfLink: /apis/config.openshift.io/v1/images/cluster
  uid: e34555da-78a9-11e9-b92b-06d6c7da38dc
spec:
  allowedRegistriesForImport: 2
    - domainName: quay.io
      insecure: false
  additionalTrustedCA: 3
    name: myconfigmap
  registrySources: 4
    allowedRegistries:
      - example.com
      - quay.io
      - registry.redhat.io
      - image-registry.openshift-image-registry.svc:5000
    insecureRegistries:
      - insecure.com
status:
  internalRegistryHostname: image-registry.openshift-image-registry.svc:5000
```

- 1 Image:** Holds cluster-wide information about how to handle images. The canonical, and only valid name is **cluster**.
- 2 allowedRegistriesForImport:** Limits the container image registries from which normal users may import images. Set this list to the registries that you trust to contain valid images, and that you want applications to be able to import from. Users with permission to create images or **ImageStreamMappings** from the API are not affected by this policy. Typically only cluster administrators have the appropriate permissions.
- 3 additionalTrustedCA:** A reference to a config map containing additional certificate authorities (CA) that are trusted during image stream import, pod image pull, **openshift-image-registry** pullthrough, and builds. The namespace for this config map is **openshift-config**. The format of the config map is to use the registry hostname as the key, and the PEM certificate as the value, for each additional registry CA to trust.

- 4 **registrySources**: Contains configuration that determines how the container runtime should treat individual registries when accessing images for builds and pods. For instance,

2. To check that the changes are applied, list your nodes:

```
$ oc get nodes
```

Example output

```

NAME                                STATUS              ROLES    AGE   VERSION
ci-ln-j5cd0qt-f76d1-vfj5x-master-0  Ready              master   98m
v1.19.0+7070803
ci-ln-j5cd0qt-f76d1-vfj5x-master-1  Ready,SchedulingDisabled  master   99m
v1.19.0+7070803
ci-ln-j5cd0qt-f76d1-vfj5x-master-2  Ready              master   98m
v1.19.0+7070803
ci-ln-j5cd0qt-f76d1-vfj5x-worker-b-nsnd4  Ready              worker   90m
v1.19.0+7070803
ci-ln-j5cd0qt-f76d1-vfj5x-worker-c-5z2gz  NotReady,SchedulingDisabled  worker   90m
v1.19.0+7070803
ci-ln-j5cd0qt-f76d1-vfj5x-worker-d-stsjv  Ready              worker   90m
v1.19.0+7070803

```

9.2.1. Adding specific registries

You can add a list of registries that are permitted for image pull and push actions by editing the **image.config.openshift.io/cluster** custom resource (CR). OpenShift Container Platform applies the changes to this CR to all nodes in the cluster.

When pulling or pushing images, the container runtime searches the registries listed under the **registrySources** parameter in the **image.config.openshift.io/cluster** CR. If you created a list of registries under the **allowedRegistries** parameter, the container runtime searches only those registries. Registries not in the list are blocked.



WARNING

When the **allowedRegistries** parameter is defined, all registries, including the **registry.redhat.io** and **quay.io** registries and the default internal image registry, are blocked unless explicitly listed. If you use the parameter, to prevent pod failure, add the **registry.redhat.io** and **quay.io** registries and the **internalRegistryHostname** to the **allowedRegistries** list, as they are required by payload images within your environment. For disconnected clusters, mirror registries should also be added.

Procedure

1. Edit the **image.config.openshift.io/cluster** CR:

```
$ oc edit image.config.openshift.io/cluster
```

The following is an example **image.config.openshift.io/cluster** CR with an allowed list:

```

apiVersion: config.openshift.io/v1
kind: Image
metadata:
  annotations:
    release.openshift.io/create-only: "true"
  creationTimestamp: "2019-05-17T13:44:26Z"
  generation: 1
  name: cluster
  resourceVersion: "8302"
  selfLink: /apis/config.openshift.io/v1/images/cluster
  uid: e34555da-78a9-11e9-b92b-06d6c7da38dc
spec:
  registrySources: 1
  allowedRegistries: 2
    - example.com
    - quay.io
    - registry.redhat.io
    - image-registry.openshift-image-registry.svc:5000
status:
  internalRegistryHostname: image-registry.openshift-image-registry.svc:5000

```

- 1 **registrySources:** Contains configurations that determine how the container runtime should treat individual registries when accessing images for builds and pods. It does not contain configuration for the internal cluster registry.
- 2 **allowedRegistries:** Registries to use for image pull and push actions. All other registries are blocked.



NOTE

Either the **allowedRegistries** parameter or the **blockedRegistries** parameter can be set, but not both.

The Machine Config Operator (MCO) watches the **image.config.openshift.io/cluster** CR for any changes to registries and reboots the nodes when it detects changes. Changes to the allowed registries creates or updates the image signature policy in the **/host/etc/containers/policy.json** file on each node.

2. To check that the registries have been added to the policy file, use the following command on a node:

```
$ cat /host/etc/containers/policy.json
```

The following policy indicates that only images from the example.com, quay.io, and registry.redhat.io registries are permitted for image pulls and pushes:

Example 9.1. Example image signature policy file

```
{
  "default": [
    {
```

```

    "type":"reject"
  }
],
"transports":{
  "atomic":{
    "example.com":[
      {
        "type":"insecureAcceptAnything"
      }
    ],
    "image-registry.openshift-image-registry.svc:5000":[
      {
        "type":"insecureAcceptAnything"
      }
    ],
    "insecure.com":[
      {
        "type":"insecureAcceptAnything"
      }
    ],
    "quay.io":[
      {
        "type":"insecureAcceptAnything"
      }
    ],
    "reg4.io/myrepo/myapp:latest":[
      {
        "type":"insecureAcceptAnything"
      }
    ],
    "registry.redhat.io":[
      {
        "type":"insecureAcceptAnything"
      }
    ]
  ],
  "docker":{
    "example.com":[
      {
        "type":"insecureAcceptAnything"
      }
    ],
    "image-registry.openshift-image-registry.svc:5000":[
      {
        "type":"insecureAcceptAnything"
      }
    ],
    "insecure.com":[
      {
        "type":"insecureAcceptAnything"
      }
    ],
    "quay.io":[
      {
        "type":"insecureAcceptAnything"
      }
    ]
  }
}

```

```

    ],
    "reg4.io/myrepo/myapp:latest":[
      {
        "type":"insecureAcceptAnything"
      }
    ],
    "registry.redhat.io":[
      {
        "type":"insecureAcceptAnything"
      }
    ]
  ],
  "docker-daemon":{
    "":[
      {
        "type":"insecureAcceptAnything"
      }
    ]
  }
}
}
}

```

NOTE

If your cluster uses the **registrySources.insecureRegistries** parameter, ensure that any insecure registries are included in the allowed list.

For example:

```

spec:
  registrySources:
    insecureRegistries:
    - insecure.com
    allowedRegistries:
    - example.com
    - quay.io
    - registry.redhat.io
    - insecure.com
    - image-registry.openshift-image-registry.svc:5000

```

9.2.2. Blocking specific registries

You can block any registry by editing the **image.config.openshift.io/cluster** custom resource (CR). OpenShift Container Platform applies the changes to this CR to all nodes in the cluster.

When pulling or pushing images, the container runtime searches the registries listed under the **registrySources** parameter in the **image.config.openshift.io/cluster** CR. If you created a list of registries under the **blockedRegistries** parameter, the container runtime does not search those registries. All other registries are allowed.

**WARNING**

To prevent pod failure, do not add the **registry.redhat.io** and **quay.io** registries to the **blockedRegistries** list, as they are required by payload images within your environment.

Procedure

1. Edit the **image.config.openshift.io/cluster** CR:

```
$ oc edit image.config.openshift.io/cluster
```

The following is an example **image.config.openshift.io/cluster** CR with a blocked list:

```
apiVersion: config.openshift.io/v1
kind: Image
metadata:
  annotations:
    release.openshift.io/create-only: "true"
    creationTimestamp: "2019-05-17T13:44:26Z"
  generation: 1
  name: cluster
  resourceVersion: "8302"
  selfLink: /apis/config.openshift.io/v1/images/cluster
  uid: e34555da-78a9-11e9-b92b-06d6c7da38dc
spec:
  registrySources: 1
  blockedRegistries: 2
  - untrusted.com
status:
  internalRegistryHostname: image-registry.openshift-image-registry.svc:5000
```

- 1 **registrySources**: Contains configurations that determine how the container runtime should treat individual registries when accessing images for builds and pods. It does not contain configuration for the internal cluster registry.
- 2 Specify registries that should not be used for image pull and push actions. All other registries are allowed.

**NOTE**

Either the **blockedRegistries** registry or the **allowedRegistries** registry can be set, but not both.

The Machine Config Operator (MCO) watches the **image.config.openshift.io/cluster** CR for any changes to registries and reboots the nodes when it detects changes. Changes to the blocked registries appear in the **/etc/containers/registries.conf** file on each node.

- To check that the registries have been added to the policy file, use the following command on a node:

```
$ cat /host/etc/containers/registries.conf
```

The following example indicates that images from the **untrusted.com** registry are prevented for image pulls and pushes:

Example output

```
unqualified-search-registries = ["registry.access.redhat.com", "docker.io"]

[[registry]]
  prefix = ""
  location = "untrusted.com"
  blocked = true
```

9.2.3. Allowing insecure registries

You can add insecure registries by editing the **image.config.openshift.io/cluster** custom resource (CR). OpenShift Container Platform applies the changes to this CR to all nodes in the cluster.

Registries that do not use valid SSL certificates or do not require HTTPS connections are considered insecure.



WARNING

Insecure external registries should be avoided to reduce possible security risks.

Procedure

- Edit the **image.config.openshift.io/cluster** CR:

```
$ oc edit image.config.openshift.io/cluster
```

The following is an example **image.config.openshift.io/cluster** CR with an insecure registries list:

```
apiVersion: config.openshift.io/v1
kind: Image
metadata:
  annotations:
    release.openshift.io/create-only: "true"
  creationTimestamp: "2019-05-17T13:44:26Z"
  generation: 1
  name: cluster
  resourceVersion: "8302"
  selfLink: /apis/config.openshift.io/v1/images/cluster
  uid: e34555da-78a9-11e9-b92b-06d6c7da38dc
```

```
spec:
  registrySources: 1
    insecureRegistries: 2
    - insecure.com
    allowedRegistries:
    - example.com
    - quay.io
    - registry.redhat.io
    - insecure.com 3
  status:
    internalRegistryHostname: image-registry.openshift-image-registry.svc:5000
```

- 1 **registrySources:** Contains configurations that determine how the container runtime should treat individual registries when accessing images for builds and pods. It does not contain configuration for the internal cluster registry.
- 2 Specify an insecure registry.
- 3 Ensure that any insecure registries are included in the **allowedRegistries** list.



NOTE

When the **allowedRegistries** parameter is defined, all registries, including the `registry.redhat.io` and `quay.io` registries and the default internal image registry, are blocked unless explicitly listed. If you use the parameter, to prevent pod failure, add all registries including the **registry.redhat.io** and **quay.io** registries and the **internalRegistryHostname** to the **allowedRegistries** list, as they are required by payload images within your environment. For disconnected clusters, mirror registries should also be added.

The Machine Config Operator (MCO) watches the **image.config.openshift.io/cluster** CR for any changes to registries and reboots the nodes when it detects changes. Changes to the insecure and blocked registries appear in the **/etc/containers/registries.conf** file on each node.

2. To check that the registries have been added to the policy file, use the following command on a node:

```
$ cat /host/etc/containers/registries.conf
```

The following example indicates that images from the **insecure.com** registry is insecure and is allowed for image pulls and pushes.

Example output

```
unqualified-search-registries = ["registry.access.redhat.com", "docker.io"]

[[registry]]
  prefix = ""
  location = "insecure.com"
  insecure = true
```

9.2.4. Configuring additional trust stores for image registry access

The **image.config.openshift.io/cluster** custom resource can contain a reference to a config map that contains additional certificate authorities to be trusted during image registry access.

Prerequisites

- The certificate authorities (CA) must be PEM-encoded.

Procedure

You can create a config map in the **openshift-config** namespace and use its name in **AdditionalTrustedCA** in the **image.config.openshift.io** custom resource to provide additional CAs that should be trusted when contacting external registries.

The config map key is the host name of a registry with the port for which this CA is to be trusted, and the base64-encoded certificate is the value, for each additional registry CA to trust.

Image registry CA config map example

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-registry-ca
data:
  registry.example.com: |
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
  registry-with-port.example.com:5000: | 1
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
```

- 1 If the registry has the port, such as **registry-with-port.example.com:5000**, **:** should be replaced with **..**

You can configure additional CAs with the following procedure.

1. To configure an additional CA:

```
$ oc create configmap registry-config --from-file=<external_registry_address>=ca.crt -n
openshift-config
```

```
$ oc edit image.config.openshift.io cluster
```

```
spec:
  additionalTrustedCA:
    name: registry-config
```

9.2.5. Configuring image registry repository mirroring

Setting up container registry repository mirroring enables you to do the following:

- Configure your OpenShift Container Platform cluster to redirect requests to pull images from a repository on a source image registry and have it resolved by a repository on a mirrored image registry.
- Identify multiple mirrored repositories for each target repository, to make sure that if one mirror is down, another can be used.

The attributes of repository mirroring in OpenShift Container Platform include:

- Image pulls are resilient to registry downtimes.
- Clusters in restricted networks can pull images from critical locations, such as quay.io, and have registries behind a company firewall provide the requested images.
- A particular order of registries is tried when an image pull request is made, with the permanent registry typically being the last one tried.
- The mirror information you enter is added to the `/etc/containers/registries.conf` file on every node in the OpenShift Container Platform cluster.
- When a node makes a request for an image from the source repository, it tries each mirrored repository in turn until it finds the requested content. If all mirrors fail, the cluster tries the source repository. If successful, the image is pulled to the node.

Setting up repository mirroring can be done in the following ways:

- At OpenShift Container Platform installation:
By pulling container images needed by OpenShift Container Platform and then bringing those images behind your company's firewall, you can install OpenShift Container Platform into a datacenter that is in a restricted network.
- After OpenShift Container Platform installation:
Even if you don't configure mirroring during OpenShift Container Platform installation, you can do so later using the **ImageContentSourcePolicy** object.

The following procedure provides a post-installation mirror configuration, where you create an **ImageContentSourcePolicy** object that identifies:

- The source of the container image repository you want to mirror.
- A separate entry for each mirror repository you want to offer the content requested from the source repository.



NOTE

You can only configure global pull secrets for clusters that have an **ImageContentSourcePolicy** object. You cannot add a pull secret to a project.

Prerequisites

- Access to the cluster as a user with the **cluster-admin** role.

Procedure

1. Configure mirrored repositories, by either:

- Setting up a mirrored repository with Red Hat Quay, as described in [Red Hat Quay Repository Mirroring](#). Using Red Hat Quay allows you to copy images from one repository to another and also automatically sync those repositories repeatedly over time.
- Using a tool such as **skopeo** to copy images manually from the source directory to the mirrored repository.
For example, after installing the skopeo RPM package on a Red Hat Enterprise Linux (RHEL) 7 or RHEL 8 system, use the **skopeo** command as shown in this example:

```
$ skopeo copy \
docker://registry.access.redhat.com/ubi8/ubi-
minimal@sha256:5cfbaf45ca96806917830c183e9f37df2e913b187adb32e89fd83fa455eba
a6 \
docker://example.io/example/ubi-minimal
```

In this example, you have a container image registry that is named **example.io** with an image repository named **example** to which you want to copy the **ubi8/ubi-minimal** image from **registry.access.redhat.com**. After you create the registry, you can configure your OpenShift Container Platform cluster to redirect requests made of the source repository to the mirrored repository.

2. Log in to your OpenShift Container Platform cluster.
3. Create an **ImageContentSourcePolicy** file (for example, **registryrepomirror.yaml**), replacing the source and mirrors with your own registry and repository pairs and images:

```
apiVersion: operator.openshift.io/v1alpha1
kind: ImageContentSourcePolicy
metadata:
  name: ubi8repo
spec:
  repositoryDigestMirrors:
  - mirrors:
    - example.io/example/ubi-minimal 1
    source: registry.access.redhat.com/ubi8/ubi-minimal 2
  - mirrors:
    - example.com/example/ubi-minimal
    source: registry.access.redhat.com/ubi8/ubi-minimal
```

- 1** Indicates the name of the image registry and repository.
- 2** Indicates the registry and repository containing the content that is mirrored.

4. Create the new **ImageContentSourcePolicy** object:

```
$ oc create -f registryrepomirror.yaml
```

After the **ImageContentSourcePolicy** object is created, the new settings are deployed to each node and the cluster starts using the mirrored repository for requests to the source repository.

5. To check that the mirrored configuration settings, are applied, do the following on one of the nodes.
 - a. List your nodes:

```
$ oc get node
```

Example output

```
NAME                                STATUS                                ROLES  AGE  VERSION
ip-10-0-137-44.ec2.internal        Ready                                worker 7m  v1.18.3
ip-10-0-138-148.ec2.internal        Ready                                master 11m v1.18.3
ip-10-0-139-122.ec2.internal        Ready                                master 11m v1.18.3
ip-10-0-147-35.ec2.internal        Ready,SchedulingDisabled            worker 7m  v1.18.3
ip-10-0-153-12.ec2.internal        Ready                                worker 7m  v1.18.3
ip-10-0-154-10.ec2.internal        Ready                                master 11m v1.18.3
```

You can see that scheduling on each worker node is disabled as the change is being applied.

- b. Start the debugging process to access the node:

```
$ oc debug node/ip-10-0-147-35.ec2.internal
```

Example output

```
Starting pod/ip-10-0-147-35ec2internal-debug ...
To use host binaries, run `chroot /host`
```

- c. Access the node's files:

```
sh-4.2# chroot /host
```

- d. Check the `/etc/containers/registries.conf` file to make sure the changes were made:

```
sh-4.2# cat /etc/containers/registries.conf
```

Example output

```
unqualified-search-registries = ["registry.access.redhat.com", "docker.io"]
[[registry]]
  location = "registry.access.redhat.com/ubi8/"
  insecure = false
  blocked = false
  mirror-by-digest-only = true
  prefix = ""

[[registry.mirror]]
  location = "example.io/example/ubi8-minimal"
  insecure = false

[[registry.mirror]]
  location = "example.com/example/ubi8-minimal"
  insecure = false
```

- e. Pull an image digest to the node from the source and check if it is resolved by the mirror. **ImageContentSourcePolicy** objects support image digests only, not image tags.

```
sh-4.2# podman pull --log-level=debug registry.access.redhat.com/ubi8/ubi-  
minimal@sha256:5cfbaf45ca96806917830c183e9f37df2e913b187adb32e89fd83fa455eba  
a6
```

Troubleshooting repository mirroring

If the repository mirroring procedure does not work as described, use the following information about how repository mirroring works to help troubleshoot the problem.

- The first working mirror is used to supply the pulled image.
- The main registry is only used if no other mirror works.
- From the system context, the **Insecure** flags are used as fallback.
- The format of the `/etc/containers/registries.conf` file has changed recently. It is now version 2 and in TOML format.

Additional resources

For more information about global pull secrets, see [Updating the global cluster pull secret](#).

CHAPTER 10. USING TEMPLATES

The following sections provide an overview of templates, as well as how to use and create them.

10.1. UNDERSTANDING TEMPLATES

A template describes a set of objects that can be parameterized and processed to produce a list of objects for creation by OpenShift Container Platform. A template can be processed to create anything you have permission to create within a project, for example services, build configurations, and deployment configurations. A template can also define a set of labels to apply to every object defined in the template.

You can create a list of objects from a template using the CLI or, if a template has been uploaded to your project or the global template library, using the web console.

10.2. UPLOADING A TEMPLATE

If you have a JSON or YAML file that defines a template, for example as seen in this example, you can upload the template to projects using the CLI. This saves the template to the project for repeated use by any user with appropriate access to that project. Instructions on writing your own templates are provided later in this topic.

Procedure

- Upload a template to your current project's template library, pass the JSON or YAML file with the following command:

```
$ oc create -f <filename>
```

- Upload a template to a different project using the **-n** option with the name of the project:

```
$ oc create -f <filename> -n <project>
```

The template is now available for selection using the web console or the CLI.

10.3. CREATING AN APPLICATION BY USING THE WEB CONSOLE

You can use the web console to create an application from a template.

Procedure

1. While in the desired project, click **Add to Project**.
2. Select either a builder image from the list of images in your project, or from the service catalog.



NOTE

Only image stream tags that have the **builder** tag listed in their annotations appear in this list, as demonstrated here:

```
kind: "ImageStream"
apiVersion: "v1"
```

```

metadata:
  name: "ruby"
  creationTimestamp: null
spec:
  dockerImageRepository: "registry.redhat.io/rhsc/ruby-26-rhel7"
  tags:
  -
    name: "2.6"
    annotations:
      description: "Build and run Ruby 2.6 applications"
      iconClass: "icon-ruby"
      tags: "builder,ruby" 1
      supports: "ruby:2.6,ruby"
      version: "2.6"

```

- 1** Including **builder** here ensures this image stream tag appears in the web console as a builder.

3. Modify the settings in the new application screen to configure the objects to support your application.

10.4. CREATING OBJECTS FROM TEMPLATES BY USING THE CLI

You can use the CLI to process templates and use the configuration that is generated to create objects.

10.4.1. Adding labels

Labels are used to manage and organize generated objects, such as pods. The labels specified in the template are applied to every object that is generated from the template.

Procedure

- Add labels in the template from the command line:

```
$ oc process -f <filename> -l name=otherLabel
```

10.4.2. Listing parameters

The list of parameters that you can override are listed in the **parameters** section of the template.

Procedure

1. You can list parameters with the CLI by using the following command and specifying the file to be used:

```
$ oc process --parameters -f <filename>
```

Alternatively, if the template is already uploaded:

```
$ oc process --parameters -n <project> <template_name>
```

For example, the following shows the output when listing the parameters for one of the Quickstart templates in the default **openshift** project:

```
$ oc process --parameters -n openshift rails-postgresql-example
```

Example output

```

NAME                DESCRIPTION
GENERATOR           VALUE
SOURCE_REPOSITORY_URL  The URL of the repository with your application source
code                https://github.com/sclorg/rails-ex.git
SOURCE_REPOSITORY_REF  Set this to a branch name, tag or other ref of your
repository if you are not using the default branch
CONTEXT_DIR          Set this to the relative path to your project if it is not in the root of
your repository
APPLICATION_DOMAIN     The exposed hostname that will route to the Rails service
rails-postgresql-example.openshiftapps.com
GITHUB_WEBHOOK_SECRET  A secret string used to configure the GitHub webhook
expression           [a-zA-Z0-9]{40}
SECRET_KEY_BASE        Your secret key for verifying the integrity of signed cookies
expression           [a-z0-9]{127}
APPLICATION_USER       The application user that is used within the sample application
to authorize access on pages                openshift
APPLICATION_PASSWORD   The application password that is used within the sample
application to authorize access on pages                secret
DATABASE_SERVICE_NAME  Database service name
postgresql
POSTGRESQL_USER        database username
expression             user[A-Z0-9]{3}
POSTGRESQL_PASSWORD    database password
expression             [a-zA-Z0-9]{8}
POSTGRESQL_DATABASE    database name
root
POSTGRESQL_MAX_CONNECTIONS  database max connections
10
POSTGRESQL_SHARED_BUFFERS  database shared buffers
12MB

```

The output identifies several parameters that are generated with a regular expression-like generator when the template is processed.

10.4.3. Generating a list of objects

Using the CLI, you can process a file defining a template to return the list of objects to standard output.

Procedure

1. Process a file defining a template to return the list of objects to standard output:

```
$ oc process -f <filename>
```

Alternatively, if the template has already been uploaded to the current project:

```
$ oc process <template_name>
```

2. Create objects from a template by processing the template and piping the output to **oc create**:

```
$ oc process -f <filename> | oc create -f -
```

Alternatively, if the template has already been uploaded to the current project:

```
$ oc process <template> | oc create -f -
```

3. You can override any parameter values defined in the file by adding the **-p** option for each **<name>=<value>** pair you want to override. A parameter reference appears in any text field inside the template items.

For example, in the following the **POSTGRESQL_USER** and **POSTGRESQL_DATABASE** parameters of a template are overridden to output a configuration with customized environment variables:

- a. Creating a List of objects from a template

```
$ oc process -f my-rails-postgresql \
  -p POSTGRESQL_USER=bob \
  -p POSTGRESQL_DATABASE=mydatabase
```

- b. The JSON file can either be redirected to a file or applied directly without uploading the template by piping the processed output to the **oc create** command:

```
$ oc process -f my-rails-postgresql \
  -p POSTGRESQL_USER=bob \
  -p POSTGRESQL_DATABASE=mydatabase \
  | oc create -f -
```

- c. If you have large number of parameters, you can store them in a file and then pass this file to **oc process**:

```
$ cat postgres.env
POSTGRESQL_USER=bob
POSTGRESQL_DATABASE=mydatabase
```

```
$ oc process -f my-rails-postgresql --param-file=postgres.env
```

- d. You can also read the environment from standard input by using **"-"** as the argument to **--param-file**:

```
$ sed s/bob/alice/ postgres.env | oc process -f my-rails-postgresql --param-file=-
```

10.5. MODIFYING UPLOADED TEMPLATES

You can edit a template that has already been uploaded to your project.

Procedure

- Modify a template that has already been uploaded:

```
$ oc edit template <template>
```

10.6. USING INSTANT APP AND QUICKSTART TEMPLATES

OpenShift Container Platform provides a number of default Instant App and Quickstart templates to make it easy to quickly get started creating a new application for different languages. Templates are provided for Rails (Ruby), Django (Python), Node.js, CakePHP (PHP), and Dancer (Perl). Your cluster administrator must create these templates in the default, global **openshift** project so you have access to them.

By default, the templates build using a public source repository on GitHub that contains the necessary application code.

Procedure

1. You can list the available default Instant App and Quickstart templates with:

```
$ oc get templates -n openshift
```

2. To modify the source and build your own version of the application:
 - a. Fork the repository referenced by the template's default **SOURCE_REPOSITORY_URL** parameter.
 - b. Override the value of the **SOURCE_REPOSITORY_URL** parameter when creating from the template, specifying your fork instead of the default value.
By doing this, the build configuration created by the template now points to your fork of the application code, and you can modify the code and rebuild the application at will.



NOTE

Some of the Instant App and Quickstart templates define a database deployment configuration. The configuration they define uses ephemeral storage for the database content. These templates should be used for demonstration purposes only as all database data is lost if the database pod restarts for any reason.

10.6.1. Quickstart templates

A Quickstart is a basic example of an application running on OpenShift Container Platform. Quickstarts come in a variety of languages and frameworks, and are defined in a template, which is constructed from a set of services, build configurations, and deployment configurations. This template references the necessary images and source repositories to build and deploy the application.

To explore a Quickstart, create an application from a template. Your administrator must have already installed these templates in your OpenShift Container Platform cluster, in which case you can simply select it from the web console.

Quickstarts refer to a source repository that contains the application source code. To customize the Quickstart, fork the repository and, when creating an application from the template, substitute the default source repository name with your forked repository. This results in builds that are performed using your source code instead of the provided example source. You can then update the code in your source repository and launch a new build to see the changes reflected in the deployed application.

10.6.1.1. Web framework Quickstart templates

These Quickstart templates provide a basic application of the indicated framework and language:

- CakePHP: a PHP web framework that includes a MySQL database
- Dancer: a Perl web framework that includes a MySQL database
- Django: a Python web framework that includes a PostgreSQL database
- NodeJS: a NodeJS web application that includes a MongoDB database
- Rails: a Ruby web framework that includes a PostgreSQL database

10.7. WRITING TEMPLATES

You can define new templates to make it easy to recreate all the objects of your application. The template defines the objects it creates along with some metadata to guide the creation of those objects.

The following is an example of a simple template object definition (YAML):

```
apiVersion: v1
kind: Template
metadata:
  name: redis-template
  annotations:
    description: "Description"
    iconClass: "icon-redis"
    tags: "database,nosql"
objects:
- apiVersion: v1
  kind: Pod
  metadata:
    name: redis-master
  spec:
    containers:
    - env:
      - name: REDIS_PASSWORD
        value: ${REDIS_PASSWORD}
      image: dockerfile/redis
      name: master
      ports:
      - containerPort: 6379
        protocol: TCP
  parameters:
  - description: Password used for Redis authentication
    from: '[A-Z0-9]{8}'
    generate: expression
    name: REDIS_PASSWORD
  labels:
    redis: master
```

10.7.1. Writing the template description

The template description informs you what the template does and helps you find it when searching in the web console. Additional metadata beyond the template name is optional, but useful to have. In addition to general descriptive information, the metadata also includes a set of tags. Useful tags include the name of the language the template is related to for example, Java, PHP, Ruby, and so on.

The following is an example of template description metadata:

```

kind: Template
apiVersion: v1
metadata:
  name: cakephp-mysql-example 1
  annotations:
    openshift.io/display-name: "CakePHP MySQL Example (Ephemeral)" 2
  description: >-
    An example CakePHP application with a MySQL database. For more information
    about using this template, including OpenShift considerations, see
    https://github.com/sclorg/cakephp-ex/blob/master/README.md.

    WARNING: Any data stored will be lost upon pod destruction. Only use this
    template for testing." 3
  openshift.io/long-description: >-
    This template defines resources needed to develop a CakePHP application,
    including a build configuration, application DeploymentConfig, and
    database DeploymentConfig. The database is stored in
    non-persistent storage, so this configuration should be used for
    experimental purposes only. 4
  tags: "quickstart,php,cakephp" 5
  iconClass: icon-php 6
  openshift.io/provider-display-name: "Red Hat, Inc." 7
  openshift.io/documentation-url: "https://github.com/sclorg/cakephp-ex" 8
  openshift.io/support-url: "https://access.redhat.com" 9
message: "Your admin credentials are ${ADMIN_USERNAME}:${ADMIN_PASSWORD}" 10

```

- 1** The unique name of the template.
- 2** A brief, user-friendly name, which can be employed by user interfaces.
- 3** A description of the template. Include enough detail that users understand what is being deployed and any caveats they must know before deploying. It should also provide links to additional information, such as a README file. Newlines can be included to create paragraphs.
- 4** Additional template description. This may be displayed by the service catalog, for example.
- 5** Tags to be associated with the template for searching and grouping. Add tags that include it into one of the provided catalog categories. Refer to the **id** and **categoryAliases** in **CATALOG_CATEGORIES** in the console constants file. The categories can also be customized for the whole cluster.
- 6** An icon to be displayed with your template in the web console.

Example 10.1. Available icons

- **icon-3scale**
- **icon-aerogear**
- **icon-amq**
- **icon-angularjs**

- **icon-ansible**
- **icon-apache**
- **icon-beaker**
- **icon-camel**
- **icon-capedwarf**
- **icon-cassandra**
- **icon-catalog-icon**
- **icon-clojure**
- **icon-codeigniter**
- **icon-cordova**
- **icon-datagrid**
- **icon-datavirt**
- **icon-debian**
- **icon-decisionserver**
- **icon-django**
- **icon-dotnet**
- **icon-drupal**
- **icon-eap**
- **icon-elastic**
- **icon-erlang**
- **icon-fedora**
- **icon-freebsd**
- **icon-git**
- **icon-github**
- **icon-gitlab**
- **icon-glassfish**
- **icon-go-gopher**
- **icon-golang**
- **icon-grails**

- **icon-hadoop**
- **icon-haproxy**
- **icon-helm**
- **icon-infinispan**
- **icon-jboss**
- **icon-jenkins**
- **icon-jetty**
- **icon-joomla**
- **icon-jruby**
- **icon-js**
- **icon-knative**
- **icon-kubevirt**
- **icon-laravel**
- **icon-load-balancer**
- **icon-mariadb**
- **icon-mediawiki**
- **icon-memcached**
- **icon-mongodb**
- **icon-mssql**
- **icon-mysql-database**
- **icon-nginx**
- **icon-nodejs**
- **icon-openjdk**
- **icon-openliberty**
- **icon-openshift**
- **icon-openstack**
- **icon-other-linux**
- **icon-other-unknown**
- **icon-perl**

- **icon-phalcon**
- **icon-php**
- **icon-play**
- **iconpostgresql**
- **icon-processserver**
- **icon-python**
- **icon-quarkus**
- **icon-rabbitmq**
- **icon-rails**
- **icon-redhat**
- **icon-redis**
- **icon-rh-integration**
- **icon-rh-spring-boot**
- **icon-rh-tomcat**
- **icon-ruby**
- **icon-scala**
- **icon-serverlessfx**
- **icon-shadowman**
- **icon-spring-boot**
- **icon-spring**
- **icon-sso**
- **icon-stackoverflow**
- **icon-suse**
- **icon-symfony**
- **icon-tomcat**
- **icon-ubuntu**
- **icon-vertx**
- **icon-wildfly**
- **icon-windows**

- **icon-wordpress**
- **icon-xamarin**
- **icon-zend**

- 7 The name of the person or organization providing the template.
- 8 A URL referencing further documentation for the template.
- 9 A URL where support can be obtained for the template.
- 10 An instructional message that is displayed when this template is instantiated. This field should inform the user how to use the newly created resources. Parameter substitution is performed on the message before being displayed so that generated credentials and other parameters can be included in the output. Include links to any next-steps documentation that users should follow.

10.7.2. Writing template labels

Templates can include a set of labels. These labels are added to each object created when the template is instantiated. Defining a label in this way makes it easy for users to find and manage all the objects created from a particular template.

The following is an example of template object labels:

```
kind: "Template"
apiVersion: "v1"
...
labels:
  template: "cakephp-mysql-example" 1
  app: "${NAME}" 2
```

- 1 A label that is applied to all objects created from this template.
- 2 A parameterized label that is also applied to all objects created from this template. Parameter expansion is carried out on both label keys and values.

10.7.3. Writing template parameters

Parameters allow a value to be supplied by you or generated when the template is instantiated. Then, that value is substituted wherever the parameter is referenced. References can be defined in any field in the objects list field. This is useful for generating random passwords or allowing you to supply a host name or other user-specific value that is required to customize the template. Parameters can be referenced in two ways:

- As a string value by placing values in the form **\${PARAMETER_NAME}** in any string field in the template.
- As a JSON or YAML value by placing values in the form **\${PARAMETER_NAME}** in place of any field in the template.

When using the **\${PARAMETER_NAME}** syntax, multiple parameter references can be combined in a

single field and the reference can be embedded within fixed data, such as `"http://${PARAMETER_1}${PARAMETER_2}"`. Both parameter values are substituted and the resulting value is a quoted string.

When using the `${PARAMETER_NAME}` syntax only a single parameter reference is allowed and leading and trailing characters are not permitted. The resulting value is unquoted unless, after substitution is performed, the result is not a valid JSON object. If the result is not a valid JSON value, the resulting value is quoted and treated as a standard string.

A single parameter can be referenced multiple times within a template and it can be referenced using both substitution syntaxes within a single template.

A default value can be provided, which is used if you do not supply a different value:

The following is an example of setting an explicit value as the default value:

```
parameters:
- name: USERNAME
  description: "The user name for Joe"
  value: joe
```

Parameter values can also be generated based on rules specified in the parameter definition, for example generating a parameter value:

```
parameters:
- name: PASSWORD
  description: "The random user password"
  generate: expression
  from: "[a-zA-Z0-9]{12}"
```

In the previous example, processing generates a random password 12 characters long consisting of all upper and lowercase alphabet letters and numbers.

The syntax available is not a full regular expression syntax. However, you can use `\w`, `\d`, `\a`, and `\A` modifiers:

- `[w]{10}` produces 10 alphabet characters, numbers, and underscores. This follows the PCRE standard and is equal to `[a-zA-Z0-9_]{10}`.
- `[d]{10}` produces 10 numbers. This is equal to `[0-9]{10}`.
- `[a]{10}` produces 10 alphabetical characters. This is equal to `[a-zA-Z]{10}`.
- `[A]{10}` produces 10 punctuation or symbol characters. This is equal to `[~!@#$$%^&*()\- _+={} \[\] \<, >, ., ?, /, ' , ; , :] {10}`.

NOTE

Depending on if the template is written in YAML or JSON, and the type of string that the modifier is embedded within, you might need to escape the backslash with a second backslash. The following examples are equivalent:

Example YAML template with a modifier

```
parameters:
- name: singlequoted_example
  generate: expression
  from: '[A]{10}'
- name: doublequoted_example
  generate: expression
  from: "[\A]{10}"
```

Example JSON template with a modifier

```
{
  "parameters": [
    {
      "name": "json_example",
      "generate": "expression",
      "from": "[\A]{10}"
    }
  ]
}
```

Here is an example of a full template with parameter definitions and references:

```
kind: Template
apiVersion: v1
metadata:
  name: my-template
objects:
- kind: BuildConfig
  apiVersion: v1
  metadata:
    name: cakephp-mysql-example
    annotations:
      description: Defines how to build the application
  spec:
    source:
      type: Git
      git:
        uri: "${SOURCE_REPOSITORY_URL}" 1
        ref: "${SOURCE_REPOSITORY_REF}"
        contextDir: "${CONTEXT_DIR}"
- kind: DeploymentConfig
  apiVersion: v1
  metadata:
    name: frontend
  spec:
    replicas: "${REPLICA_COUNT}" 2
parameters:
```

```

- name: SOURCE_REPOSITORY_URL 3
  displayName: Source Repository URL 4
  description: The URL of the repository with your application source code 5
  value: https://github.com/sclorg/cakephp-ex.git 6
  required: true 7
- name: GITHUB_WEBHOOK_SECRET
  description: A secret string used to configure the GitHub webhook
  generate: expression 8
  from: "[a-zA-Z0-9]{40}" 9
- name: REPLICAS_COUNT
  description: Number of replicas to run
  value: "2"
  required: true
message: "... The GitHub webhook secret is ${GITHUB_WEBHOOK_SECRET} ..." 10

```

- 1** This value is replaced with the value of the **SOURCE_REPOSITORY_URL** parameter when the template is instantiated.
- 2** This value is replaced with the unquoted value of the **REPLICAS_COUNT** parameter when the template is instantiated.
- 3** The name of the parameter. This value is used to reference the parameter within the template.
- 4** The user-friendly name for the parameter. This is displayed to users.
- 5** A description of the parameter. Provide more detailed information for the purpose of the parameter, including any constraints on the expected value. Descriptions should use complete sentences to follow the console's text standards. Do not make this a duplicate of the display name.
- 6** A default value for the parameter which is used if you do not override the value when instantiating the template. Avoid using default values for things like passwords, instead use generated parameters in combination with secrets.
- 7** Indicates this parameter is required, meaning you cannot override it with an empty value. If the parameter does not provide a default or generated value, you must supply a value.
- 8** A parameter which has its value generated.
- 9** The input to the generator. In this case, the generator produces a 40 character alphanumeric value including upper and lowercase characters.
- 10** Parameters can be included in the template message. This informs you about generated values.

10.7.4. Writing the template object list

The main portion of the template is the list of objects which is created when the template is instantiated. This can be any valid API object, such as a build configuration, deployment configuration, or service. The object is created exactly as defined here, with any parameter values substituted in prior to creation. The definition of these objects can reference parameters defined earlier.

The following is an example of an object list:

```

kind: "Template"
apiVersion: "v1"

```

```

metadata:
  name: my-template
objects:
- kind: "Service" 1
  apiVersion: "v1"
  metadata:
    name: "cakephp-mysql-example"
    annotations:
      description: "Exposes and load balances the application pods"
  spec:
    ports:
      - name: "web"
        port: 8080
        targetPort: 8080
    selector:
      name: "cakephp-mysql-example"

```

- 1 The definition of a service, which is created by this template.



NOTE

If an object definition metadata includes a fixed **namespace** field value, the field is stripped out of the definition during template instantiation. If the **namespace** field contains a parameter reference, normal parameter substitution is performed and the object is created in whatever namespace the parameter substitution resolved the value to, assuming the user has permission to create objects in that namespace.

10.7.5. Marking a template as bindable

The Template Service Broker advertises one service in its catalog for each template object of which it is aware. By default, each of these services is advertised as being bindable, meaning an end user is permitted to bind against the provisioned service.

Procedure

Template authors can prevent end users from binding against services provisioned from a given template.

- Prevent end user from binding against services provisioned from a given template by adding the annotation **template.openshift.io/bindable: "false"** to the template.

10.7.6. Exposing template object fields

Template authors can indicate that fields of particular objects in a template should be exposed. The Template Service Broker recognizes exposed fields on **ConfigMap**, **Secret**, **Service**, and **Route** objects, and returns the values of the exposed fields when a user binds a service backed by the broker.

To expose one or more fields of an object, add annotations prefixed by **template.openshift.io/expose-** or **template.openshift.io/base64-expose-** to the object in the template.

Each annotation key, with its prefix removed, is passed through to become a key in a **bind** response.

Each annotation value is a Kubernetes JSONPath expression, which is resolved at bind time to indicate the object field whose value should be returned in the **bind** response.

**NOTE**

Bind response key-value pairs can be used in other parts of the system as environment variables. Therefore, it is recommended that every annotation key with its prefix removed should be a valid environment variable name – beginning with a character **A-Z**, **a-z**, or **_**, and being followed by zero or more characters **A-Z**, **a-z**, **0-9**, or **_**.

**NOTE**

Unless escaped with a backslash, Kubernetes' JSONPath implementation interprets characters such as **.**, **@**, and others as metacharacters, regardless of their position in the expression. Therefore, for example, to refer to a **ConfigMap** datum named **my.key**, the required JSONPath expression would be `{.data['my\\.key']}`. Depending on how the JSONPath expression is then written in YAML, an additional backslash might be required, for example `"{.data['my\\.\\.key']}`".

The following is an example of different objects' fields being exposed:

```
kind: Template
apiVersion: v1
metadata:
  name: my-template
objects:
- kind: ConfigMap
  apiVersion: v1
  metadata:
    name: my-template-config
    annotations:
      template.openshift.io/expose-username: "{.data['my\\.username']}"
  data:
    my.username: foo
- kind: Secret
  apiVersion: v1
  metadata:
    name: my-template-config-secret
    annotations:
      template.openshift.io/base64-expose-password: "{.data['password']}"
  stringData:
    password: bar
- kind: Service
  apiVersion: v1
  metadata:
    name: my-template-service
    annotations:
      template.openshift.io/expose-service_ip_port: "{.spec.clusterIP}:{.spec.ports[?
(.name==\"web\").port]}"
  spec:
    ports:
      - name: "web"
        port: 8080
- kind: Route
  apiVersion: v1
  metadata:
    name: my-template-route
    annotations:
```



```
template.openshift.io/expose-uri: "http://{.spec.host}{.spec.path}"
spec:
  path: mypath
```

An example response to a **bind** operation given the above partial template follows:

```
{
  "credentials": {
    "username": "foo",
    "password": "YmFy",
    "service_ip_port": "172.30.12.34:8080",
    "uri": "http://route-test.router.default.svc.cluster.local/mypath"
  }
}
```

Procedure

- Use the **template.openshift.io/expose-** annotation to return the field value as a string. This is convenient, although it does not handle arbitrary binary data.
- If you want to return binary data, use the **template.openshift.io/base64-expose-** annotation instead to base64 encode the data before it is returned.

10.7.7. Waiting for template readiness

Template authors can indicate that certain objects within a template should be waited for before a template instantiation by the service catalog, Template Service Broker, or **TemplateInstance** API is considered complete.

To use this feature, mark one or more objects of kind **Build**, **BuildConfig**, **Deployment**, **DeploymentConfig**, **Job**, or **StatefulSet** in a template with the following annotation:

```
"template.alpha.openshift.io/wait-for-ready": "true"
```

Template instantiation is not complete until all objects marked with the annotation report ready. Similarly, if any of the annotated objects report failed, or if the template fails to become ready within a fixed timeout of one hour, the template instantiation fails.

For the purposes of instantiation, readiness and failure of each object kind are defined as follows:

Kind	Readiness	Failure
Build	Object reports phase complete.	Object reports phase canceled, error, or failed.
BuildConfig	Latest associated build object reports phase complete.	Latest associated build object reports phase canceled, error, or failed.
Deployment	Object reports new replica set and deployment available. This honors readiness probes defined on the object.	Object reports progressing condition as false.

Kind	Readiness	Failure
DeploymentConfig	Object reports new replication controller and deployment available. This honors readiness probes defined on the object.	Object reports progressing condition as false.
Job	Object reports completion.	Object reports that one or more failures have occurred.
StatefulSet	Object reports all replicas ready. This honors readiness probes defined on the object.	Not applicable.

The following is an example template extract, which uses the **wait-for-ready** annotation. Further examples can be found in the OpenShift Container Platform quickstart templates.

```

kind: Template
apiVersion: v1
metadata:
  name: my-template
objects:
- kind: BuildConfig
  apiVersion: v1
  metadata:
    name: ...
    annotations:
      # wait-for-ready used on BuildConfig ensures that template instantiation
      # will fail immediately if build fails
      template.alpha.openshift.io/wait-for-ready: "true"
  spec:
    ...
- kind: DeploymentConfig
  apiVersion: v1
  metadata:
    name: ...
    annotations:
      template.alpha.openshift.io/wait-for-ready: "true"
  spec:
    ...
- kind: Service
  apiVersion: v1
  metadata:
    name: ...
  spec:
    ...

```

Additional recommendations

- Set memory, CPU, and storage default sizes to make sure your application is given enough resources to run smoothly.

- Avoid referencing the **latest** tag from images if that tag is used across major versions. This can cause running applications to break when new images are pushed to that tag.
- A good template builds and deploys cleanly without requiring modifications after the template is deployed.

10.7.8. Creating a template from existing objects

Rather than writing an entire template from scratch, you can export existing objects from your project in YAML form, and then modify the YAML from there by adding parameters and other customizations as template form.

Procedure

- Export objects in a project in YAML form:

```
$ oc get -o yaml --export all > <yaml_filename>
```

You can also substitute a particular resource type or multiple resources instead of **all**. Run **oc get -h** for more examples.

The object types included in **oc get --export all** are:

- **BuildConfig**
- **Build**
- **DeploymentConfig**
- **ImageStream**
- **Pod**
- **ReplicationController**
- **Route**
- **Service**

CHAPTER 11. USING RUBY ON RAILS

Ruby on Rails is a web framework written in Ruby. This guide covers using Rails 4 on OpenShift Container Platform.



WARNING

Go through the whole tutorial to have an overview of all the steps necessary to run your application on the OpenShift Container Platform. If you experience a problem try reading through the entire tutorial and then going back to your issue. It can also be useful to review your previous steps to ensure that all the steps were run correctly.

11.1. PREREQUISITES

- Basic Ruby and Rails knowledge.
- Locally installed version of Ruby 2.0.0+, Rubygems, Bundler.
- Basic Git knowledge.
- Running instance of OpenShift Container Platform 4.
- Make sure that an instance of OpenShift Container Platform is running and is available. Also make sure that your **oc** CLI client is installed and the command is accessible from your command shell, so you can use it to log in using your email address and password.

11.2. SETTING UP THE DATABASE

Rails applications are almost always used with a database. For local development use the PostgreSQL database.

Procedure

1. Install the database:

```
$ sudo yum install -y postgresql postgresql-server postgresql-devel
```

2. Initialize the database:

```
$ sudo postgresql-setup initdb
```

This command creates the **/var/lib/pgsql/data** directory, in which the data is stored.

3. Start the database:

```
$ sudo systemctl start postgresql.service
```

4. When the database is running, create your **rails** user:

```
$ sudo -u postgres createuser -s rails
```

Note that the user created has no password.

11.3. WRITING YOUR APPLICATION

If you are starting your Rails application from scratch, you must install the Rails gem first. Then you can proceed with writing your application.

Procedure

1. Install the Rails gem:

```
$ gem install rails
```

Example output

```
Successfully installed rails-4.3.0
1 gem installed
```

2. After you install the Rails gem, create a new application with PostgreSQL as your database:

```
$ rails new rails-app --database=postgresql
```

3. Change into your new application directory:

```
$ cd rails-app
```

4. If you already have an application, make sure the **pg** (postgresql) gem is present in your **Gemfile**. If not, edit your **Gemfile** by adding the gem:

```
gem 'pg'
```

5. Generate a new **Gemfile.lock** with all your dependencies:

```
$ bundle install
```

6. In addition to using the **postgresql** database with the **pg** gem, you also must ensure that the **config/database.yml** is using the **postgresql** adapter.

Make sure you updated **default** section in the **config/database.yml** file, so it looks like this:

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: 5
  host: localhost
  username: rails
  password:
```

7. Create your application's development and test databases:

```
$ rake db:create
```

This creates **development** and **test** database in your PostgreSQL server.

11.3.1. Creating a welcome page

Since Rails 4 no longer serves a static **public/index.html** page in production, you must create a new root page.

In order to have a custom welcome page must do following steps:

- Create a controller with an index action.
- Create a view page for the welcome controller index action.
- Create a route that serves applications root page with the created controller and view.

Rails offers a generator that completes all necessary steps for you.

Procedure

1. Run Rails generator:

```
$ rails generate controller welcome index
```

All the necessary files are created.

2. edit line 2 in **config/routes.rb** file as follows:

```
root 'welcome#index'
```

3. Run the rails server to verify the page is available:

```
$ rails server
```

You should see your page by visiting <http://localhost:3000> in your browser. If you do not see the page, check the logs that are output to your server to debug.

11.3.2. Configuring application for OpenShift Container Platform

To have your application communicate with the PostgreSQL database service running in OpenShift Container Platform you must edit the **default** section in your **config/database.yml** to use environment variables, which you must define later, upon the database service creation.

Procedure

- Edit the **default** section in your **config/database.yml** with pre-defined variables as follows:

Sample config/database YAML file

```
<% user = ENV.key?("POSTGRESQL_ADMIN_PASSWORD") ? "root" :  
ENV["POSTGRESQL_USER"] %>  
<% password = ENV.key?("POSTGRESQL_ADMIN_PASSWORD") ?  
ENV["POSTGRESQL_ADMIN_PASSWORD"] : ENV["POSTGRESQL_PASSWORD"] %>
```

```

<% db_service = ENV.fetch("DATABASE_SERVICE_NAME", "").upcase %>

default: &default
adapter: postgresql
encoding: unicode
# For details on connection pooling, see rails configuration guide
# http://guides.rubyonrails.org/configuring.html#database-pooling
pool: <%= ENV["POSTGRESQL_MAX_CONNECTIONS"] || 5 %>
username: <%= user %>
password: <%= password %>
host: <%= ENV["#{db_service}_SERVICE_HOST"] %>
port: <%= ENV["#{db_service}_SERVICE_PORT"] %>
database: <%= ENV["POSTGRESQL_DATABASE"] %>

```

11.3.3. Storing your application in Git

Building an application in OpenShift Container Platform usually requires that the source code be stored in a git repository, so you must install **git** if you do not already have it.

Prerequisites

- Install git.

Procedure

1. Make sure you are in your Rails application directory by running the **ls -1** command. The output of the command should look like:

```
$ ls -1
```

Example output

```

app
bin
config
config.ru
db
Gemfile
Gemfile.lock
lib
log
public
Rakefile
README.rdoc
test
tmp
vendor

```

2. Run the following commands in your Rails app directory to initialize and commit your code to git:

```
$ git init
```

```
$ git add .
```

```
$ git commit -m "initial commit"
```

After your application is committed you must push it to a remote repository. GitHub account, in which you create a new repository.

3. Set the remote that points to your **git** repository:

```
$ git remote add origin git@github.com:<namespace/repository-name>.git
```

4. Push your application to your remote git repository.

```
$ git push
```

11.4. DEPLOYING YOUR APPLICATION TO OPENSIFT CONTAINER PLATFORM

You can deploy you application to OpenShift Container Platform.

After creating the **rails-app** project, you are automatically switched to the new project namespace.

Deploying your application in OpenShift Container Platform involves three steps:

- Creating a database service from OpenShift Container Platform's PostgreSQL image.
- Creating a frontend service from OpenShift Container Platform's Ruby 2.0 builder image and your Ruby on Rails source code, which are wired with the database service.
- Creating a route for your application.

Procedure

- To deploy your Ruby on Rails application, create a new project for the application:

```
$ oc new-project rails-app --description="My Rails application" --display-name="Rails Application"
```

11.4.1. Creating the database service

Your Rails application expects a running database service. For this service use PostgreSQL database image.

To create the database service, use the **oc new-app** command. To this command you must pass some necessary environment variables which are used inside the database container. These environment variables are required to set the username, password, and name of the database. You can change the values of these environment variables to anything you would like. The variables are as follows:

- POSTGRESQL_DATABASE
- POSTGRESQL_USER
- POSTGRESQL_PASSWORD

Setting these variables ensures:

- A database exists with the specified name.
- A user exists with the specified name.
- The user can access the specified database with the specified password.

Procedure

1. Create the database service:

```
$ oc new-app postgresql -e POSTGRESQL_DATABASE=db_name -e
  POSTGRESQL_USER=username -e POSTGRESQL_PASSWORD=password
```

To also set the password for the database administrator, append to the previous command with:

```
-e POSTGRESQL_ADMIN_PASSWORD=admin_pw
```

2. Watch the progress:

```
$ oc get pods --watch
```

11.4.2. Creating the frontend service

To bring your application to OpenShift Container Platform, you must specify a repository in which your application lives.

Procedure

1. Create the frontend service and specify database related environment variables that were setup when creating the database service:

```
$ oc new-app path/to/source/code --name=rails-app -e POSTGRESQL_USER=username -e
  POSTGRESQL_PASSWORD=password -e POSTGRESQL_DATABASE=db_name -e
  DATABASE_SERVICE_NAME=postgresql
```

With this command, OpenShift Container Platform fetches the source code, sets up the builder builds your application image, and deploys the newly created image together with the specified environment variables. The application is named **rails-app**.

2. Verify the environment variables have been added by viewing the JSON document of the **rails-app** deployment config:

```
$ oc get dc rails-app -o json
```

You should see the following section:

3. Example output

```
env": [
  {
    "name": "POSTGRESQL_USER",
    "value": "username"
  },

```

```
[
  {
    "name": "POSTGRESQL_PASSWORD",
    "value": "password"
  },
  {
    "name": "POSTGRESQL_DATABASE",
    "value": "db_name"
  },
  {
    "name": "DATABASE_SERVICE_NAME",
    "value": "postgresql"
  }
],
```

1. Check the build process:

```
$ oc logs -f build/rails-app-1
```

2. Once the build is complete, look at the running pods in OpenShift Container Platform:

```
$ oc get pods
```

You should see a line starting with **myapp-<number>-<hash>**, and that is your application running in OpenShift Container Platform.

3. Before your application is functional, you must initialize the database by running the database migration script. There are two ways you can do this:

- Manually from the running frontend container:
 - Exec into frontend container with **rsh** command:

```
$ oc rsh <frontend_pod_id>
```

- Run the migration from inside the container:

```
$ RAILS_ENV=production bundle exec rake db:migrate
```

If you are running your Rails application in a **development** or **test** environment you do not have to specify the **RAILS_ENV** environment variable.

- By adding pre-deployment lifecycle hooks in your template.

11.4.3. Creating a route for your application

You can expose a service to create a route for your application.

Procedure

- To expose a service by giving it an externally-reachable hostname like **www.example.com** use OpenShift Container Platform route. In your case you need to expose the frontend service by typing:

```
$ oc expose service rails-app --hostname=www.example.com
```

**WARNING**

Ensure the hostname you specify resolves into the IP address of the router.

CHAPTER 12. USING IMAGES

12.1. USING IMAGES OVERVIEW

Use the following topics to discover the different Source-to-Image (S2I), database, and other container images that are available for OpenShift Container Platform users.

Red Hat official container images are provided in the Red Hat Registry at registry.redhat.io. OpenShift Container Platform's supported S2I, database, and Jenkins images are provided in the **openshift4** repository in the Red Hat Quay Registry. For example, **quay.io/openshift-release-dev/ocp-v4.0-[<address>](#)** is the name of the OpenShift Application Platform image.

The xPaaS middleware images are provided in their respective product repositories on the Red Hat Registry but suffixed with a **-openshift**. For example, **registry.redhat.io/jboss-eap-6/eap64-openshift** is the name of the JBoss EAP image.

All Red Hat supported images covered in this section are described in the [Container images section of the Red Hat Ecosystem Catalog](#). For every version of each image, you can find details on its contents and usage. Browse or search for the image that interests you.



IMPORTANT

The newer versions of container images are not compatible with earlier versions of OpenShift Container Platform. Verify and use the correct version of container images, based on your version of OpenShift Container Platform.

12.2. CONFIGURING JENKINS IMAGES

OpenShift Container Platform provides a container image for running Jenkins. This image provides a Jenkins server instance, which can be used to set up a basic flow for continuous testing, integration, and delivery.

The image is based on the Red Hat Universal Base Images (UBI).

OpenShift Container Platform follows the [LTS](#) release of Jenkins. OpenShift Container Platform provides an image that contains Jenkins 2.x.

The OpenShift Container Platform Jenkins images are available on [Quay.io](https://quay.io) or registry.redhat.io.

For example:

```
$ podman pull registry.redhat.io/openshift4/ose-jenkins:<v4.3.0>
```

To use these images, you can either access them directly from these registries or push them into your OpenShift Container Platform container image registry. Additionally, you can create an image stream that points to the image, either in your container image registry or at the external location. Your OpenShift Container Platform resources can then reference the image stream.

But for convenience, OpenShift Container Platform provides image streams in the **openshift** namespace for the core Jenkins image as well as the example Agent images provided for OpenShift Container Platform integration with Jenkins.

12.2.1. Configuration and customization

You can manage Jenkins authentication in two ways:

- OpenShift Container Platform OAuth authentication provided by the OpenShift Container Platform Login plug-in.
- Standard authentication provided by Jenkins.

12.2.1.1. OpenShift Container Platform OAuth authentication

OAuth authentication is activated by configuring options on the **Configure Global Security** panel in the Jenkins UI, or by setting the **OPENSIFT_ENABLE_OAUTH** environment variable on the Jenkins **Deployment configuration** to anything other than **false**. This activates the OpenShift Container Platform Login plug-in, which retrieves the configuration information from pod data or by interacting with the OpenShift Container Platform API server.

Valid credentials are controlled by the OpenShift Container Platform identity provider.

Jenkins supports both browser and non-browser access.

Valid users are automatically added to the Jenkins authorization matrix at log in, where OpenShift Container Platform roles dictate the specific Jenkins permissions that users have. The roles used by default are the predefined **admin**, **edit**, and **view**. The login plug-in executes self-SAR requests against those roles in the project or namespace that Jenkins is running in.

Users with the **admin** role have the traditional Jenkins administrative user permissions. Users with the **edit** or **view** role have progressively fewer permissions.

The default OpenShift Container Platform **admin**, **edit**, and **view** roles and the Jenkins permissions those roles are assigned in the Jenkins instance are configurable.

When running Jenkins in an OpenShift Container Platform pod, the login plug-in looks for a config map named **openshift-jenkins-login-plugin-config** in the namespace that Jenkins is running in.

If this plugin finds and can read in that config map, you can define the role to Jenkins Permission mappings. Specifically:

- The login plug-in treats the key and value pairs in the config map as Jenkins permission to OpenShift Container Platform role mappings.
- The key is the Jenkins permission group short ID and the Jenkins permission short ID, with those two separated by a hyphen character.
- If you want to add the **Overall Jenkins Administer** permission to an OpenShift Container Platform role, the key should be **Overall-Administer**.
- To get a sense of which permission groups and permissions IDs are available, go to the matrix authorization page in the Jenkins console and IDs for the groups and individual permissions in the table they provide.
- The value of the key and value pair is the list of OpenShift Container Platform roles the permission should apply to, with each role separated by a comma.
- If you want to add the **Overall Jenkins Administer** permission to both the default **admin** and **edit** roles, as well as a new Jenkins role you have created, the value for the key **Overall-Administer** would be **admin,edit,jenkins**.



NOTE

The **admin** user that is pre-populated in the OpenShift Container Platform Jenkins image with administrative privileges is not given those privileges when OpenShift Container Platform OAuth is used. To grant these permissions the OpenShift Container Platform cluster administrator must explicitly define that user in the OpenShift Container Platform identity provider and assigns the **admin** role to the user.

Jenkins users' permissions that are stored can be changed after the users are initially established. The OpenShift Container Platform Login plug-in polls the OpenShift Container Platform API server for permissions and updates the permissions stored in Jenkins for each user with the permissions retrieved from OpenShift Container Platform. If the Jenkins UI is used to update permissions for a Jenkins user, the permission changes are overwritten the next time the plug-in polls OpenShift Container Platform.

You can control how often the polling occurs with the **OPENSIFT_PERMISSIONS_POLL_INTERVAL** environment variable. The default polling interval is five minutes.

The easiest way to create a new Jenkins service using OAuth authentication is to use a template.

12.2.1.2. Jenkins authentication

Jenkins authentication is used by default if the image is run directly, without using a template.

The first time Jenkins starts, the configuration is created along with the administrator user and password. The default user credentials are **admin** and **password**. Configure the default password by setting the **JENKINS_PASSWORD** environment variable when using, and only when using, standard Jenkins authentication.

Procedure

- Create a Jenkins application that uses standard Jenkins authentication:

```
$ oc new-app -e \
  JENKINS_PASSWORD=<password> \
  openshift4/ose-jenkins
```

12.2.2. Jenkins environment variables

The Jenkins server can be configured with the following environment variables:

Variable	Definition	Example values and settings
OPENSIFT_ENABLE_OAUTH	Determines whether the OpenShift Container Platform Login plug-in manages authentication when logging in to Jenkins. To enable, set to true .	Default: false

Variable	Definition	Example values and settings
JENKINS_PASSWORD	The password for the admin user when using standard Jenkins authentication. Not applicable when OPENSIFT_ENABLE_OAUTH is set to true .	Default: password
JAVA_MAX_HEAP_PARAM , CONTAINER_HEAP_PERCENT , JENKINS_MAX_HEAP_UPPER_BOUND_MB	These values control the maximum heap size of the Jenkins JVM. If JAVA_MAX_HEAP_PARAM is set, its value takes precedence. Otherwise, the maximum heap size is dynamically calculated as CONTAINER_HEAP_PERCENT of the container memory limit, optionally capped at JENKINS_MAX_HEAP_UPPER_BOUND_MB MiB. By default, the maximum heap size of the Jenkins JVM is set to 50% of the container memory limit with no cap.	JAVA_MAX_HEAP_PARAM example setting: -Xmx512m CONTAINER_HEAP_PERCENT default: 0.5 , or 50% JENKINS_MAX_HEAP_UPPER_BOUND_MB example setting: 512 MiB
JAVA_INITIAL_HEAP_PARAM , CONTAINER_INITIAL_PERCENT	These values control the initial heap size of the Jenkins JVM. If JAVA_INITIAL_HEAP_PARAM is set, its value takes precedence. Otherwise, the initial heap size is dynamically calculated as CONTAINER_INITIAL_PERCENT of the dynamically calculated maximum heap size. By default, the JVM sets the initial heap size.	JAVA_INITIAL_HEAP_PARAM example setting: -Xms32m CONTAINER_INITIAL_PERCENT example setting: 0.1 , or 10%
CONTAINER_CORE_LIMIT	If set, specifies an integer number of cores used for sizing numbers of internal JVM threads.	Example setting: 2
JAVA_TOOL_OPTIONS	Specifies options to apply to all JVMs running in this container. It is not recommended to override this value.	Default: - XX:+UnlockExperimentalVMOptions - XX:+UseCGroupMemoryLimitForHeap - Dsun.zip.disableMemoryMapping=true

Variable	Definition	Example values and settings
JAVA_GC_OPTS	Specifies Jenkins JVM garbage collection parameters. It is not recommended to override this value.	Default: -XX:+UseParallelGC -XX:MinHeapFreeRatio=5 -XX:MaxHeapFreeRatio=10 -XX:GCTimeRatio=4 -XX:AdaptiveSizePolicyWeight=90
JENKINS_JAVA_OVERRIDES	Specifies additional options for the Jenkins JVM. These options are appended to all other options, including the Java options above, and may be used to override any of them if necessary. Separate each additional option with a space; if any option contains space characters, escape them with a backslash.	Example settings: -Dfoo -Dbar; -Dfoo=first\ value -Dbar=second\ value.
JENKINS_OPTS	Specifies arguments to Jenkins.	
INSTALL_PLUGINS	Specifies additional Jenkins plug-ins to install when the container is first run or when OVERVERRIDE_PV_PLUGINS_WITH_IMAGE_PLUGINS is set to true . Plug-ins are specified as a comma-delimited list of name:version pairs.	Example setting: git:3.7.0,subversion:2.10.2.
OPENSIFT_PERMISSIONS_POLL_INTERVAL	Specifies the interval in milliseconds that the OpenShift Container Platform Login plug-in polls OpenShift Container Platform for the permissions that are associated with each user that is defined in Jenkins.	Default: 300000 - 5 minutes

Variable	Definition	Example values and settings
OVERRIDE_PV_CONFIG_WITH_IMAGE_CONFIG	When running this image with an OpenShift Container Platform persistent volume (PV) for the Jenkins configuration directory, the transfer of configuration from the image to the PV is performed only the first time the image starts because the PV is assigned when the persistent volume claim (PVC) is created. If you create a custom image that extends this image and updates configuration in the custom image after the initial start-up, the configuration is not copied over unless you set this environment variable to true .	Default: false
OVERRIDE_PV_PLUGINS_WITH_IMAGE_PLUGINS	When running this image with an OpenShift Container Platform PV for the Jenkins configuration directory, the transfer of plug-ins from the image to the PV is performed only the first time the image starts because the PV is assigned when the PVC is created. If you create a custom image that extends this image and updates plug-ins in the custom image after the initial startup, the plug-ins are not copied over unless you set this environment variable to true .	Default: false
ENABLE_FATAL_ERROR_LOG_FILE	When running this image with an OpenShift Container Platform PVC for the Jenkins configuration directory, this environment variable allows the fatal error log file to persist when a fatal error occurs. The fatal error file is saved at /var/lib/jenkins/logs .	Default: false
NODEJS_SLAVE_IMAGE	Setting this value overrides the image that is used for the default Node.js agent pod configuration. A related image stream tag named jenkins-agent-nodejs is in the project. This variable must be set before Jenkins starts the first time for it to have an effect.	Default Node.js agent image in Jenkins server: image-registry.openshift-image-registry.svc:5000/openshift/jenkins-agent-nodejs:latest

Variable	Definition	Example values and settings
MAVEN_SLAVE_IMAGE	Setting this value overrides the image used for the default maven agent pod configuration. A related image stream tag named jenkins-agent-maven is in the project. This variable must be set before Jenkins starts the first time for it to have an effect.	Default Maven agent image in Jenkins server: image-registry.openshift-image-registry.svc:5000/openshift/jenkins-agent-maven:latest

12.2.3. Providing Jenkins cross project access

If you are going to run Jenkins somewhere other than your same project, you must provide an access token to Jenkins to access your project.

Procedure

1. Identify the secret for the service account that has appropriate permissions to access the project Jenkins must access:

```
$ oc describe serviceaccount jenkins
```

Example output

```
Name:      default
Labels:    <none>
Secrets:   { jenkins-token-uyswp  }
           { jenkins-dockercfg-xcr3d  }
Tokens:    jenkins-token-izv1u
           jenkins-token-uyswp
```

In this case the secret is named **jenkins-token-uyswp**.

2. Retrieve the token from the secret:

```
$ oc describe secret <secret name from above>
```

Example output

```
Name:      jenkins-token-uyswp
Labels:    <none>
Annotations:  kubernetes.io/service-account.name=jenkins,kubernetes.io/service-account.uid=32f5b661-2a8f-11e5-9528-3c970e3bf0b7
Type:       kubernetes.io/service-account-token
Data
====
ca.crt: 1066 bytes
token: eyJhbGc..<content cut>...wRA
```

The token parameter contains the token value Jenkins requires to access the project.

12.2.4. Jenkins cross volume mount points

The Jenkins image can be run with mounted volumes to enable persistent storage for the configuration:

- **/var/lib/jenkins** is the data directory where Jenkins stores configuration files, including job definitions.

12.2.5. Customizing the Jenkins image through source-to-image

To customize the official OpenShift Container Platform Jenkins image, you can use the image as a source-to-image (S2I) builder.

You can use S2I to copy your custom Jenkins jobs definitions, add additional plug-ins, or replace the provided **config.xml** file with your own, custom, configuration.

To include your modifications in the Jenkins image, you must have a Git repository with the following directory structure:

plugins

This directory contains those binary Jenkins plug-ins you want to copy into Jenkins.

plugins.txt

This file lists the plug-ins you want to install using the following syntax:

```
pluginId:pluginVersion
```

configuration/jobs

This directory contains the Jenkins job definitions.

configuration/config.xml

This file contains your custom Jenkins configuration.

The contents of the **configuration/** directory is copied to the **/var/lib/jenkins/** directory, so you can also include additional files, such as **credentials.xml**, there.

Sample build configuration customizes the Jenkins image in OpenShift Container Platform

```
apiVersion: v1
kind: BuildConfig
metadata:
  name: custom-jenkins-build
spec:
  source: 1
  git:
    uri: https://github.com/custom/repository
    type: Git
  strategy: 2
  sourceStrategy:
    from:
      kind: ImageStreamTag
      name: jenkins:2
      namespace: openshift
    type: Source
  output: 3
```

```
to:  
  kind: ImageStreamTag  
  name: custom-jenkins:latest
```

- 1 The **source** parameter defines the source Git repository with the layout described above.
- 2 The **strategy** parameter defines the original Jenkins image to use as a source image for the build.
- 3 The **output** parameter defines the resulting, customized Jenkins image that you can use in deployment configurations instead of the official Jenkins image.

12.2.6. Configuring the Jenkins Kubernetes plug-in

The OpenShift Container Platform Jenkins image includes the pre-installed [Kubernetes plug-in](#) that allows Jenkins agents to be dynamically provisioned on multiple container hosts using Kubernetes and OpenShift Container Platform.

To use the Kubernetes plug-in, OpenShift Container Platform provides images that are suitable for use as Jenkins agents including the Base, Maven, and Node.js images.

Both the Maven and Node.js agent images are automatically configured as Kubernetes pod template images within the OpenShift Container Platform Jenkins image configuration for the Kubernetes plug-in. That configuration includes labels for each of the images that can be applied to any of your Jenkins jobs under their Restrict where this project can be run setting. If the label is applied, jobs run under an OpenShift Container Platform pod running the respective agent image.

The Jenkins image also provides auto-discovery and auto-configuration of additional agent images for the Kubernetes plug-in.

With the OpenShift Container Platform sync plug-in, the Jenkins image on Jenkins start-up searches for the following within the project that it is running or the projects specifically listed in the plug-in's configuration:

- Image streams that have the label **role** set to **jenkins-agent**.
- Image stream tags that have the annotation **role** set to **jenkins-agent**.
- Config maps that have the label **role** set to **jenkins-agent**.

When it finds an image stream with the appropriate label, or image stream tag with the appropriate annotation, it generates the corresponding Kubernetes plug-in configuration so you can assign your Jenkins jobs to run in a pod that runs the container image that is provided by the image stream.

The name and image references of the image stream or image stream tag are mapped to the name and image fields in the Kubernetes plug-in pod template. You can control the label field of the Kubernetes plug-in pod template by setting an annotation on the image stream or image stream tag object with the key **agent-label**. Otherwise, the name is used as the label.

**NOTE**

Do not log in to the Jenkins console and modify the pod template configuration. If you do so after the pod template is created, and the OpenShift Container Platform Sync plug-in detects that the image associated with the image stream or image stream tag has changed, it replaces the pod template and overwrites those configuration changes. You cannot merge a new configuration with the existing configuration.

Consider the config map approach if you have more complex configuration needs.

When it finds a config map with the appropriate label, it assumes that any values in the key-value data payload of the config map contains Extensible Markup Language (XML) that is consistent with the configuration format for Jenkins and the Kubernetes plug-in pod templates. A key differentiator to note when using config maps, instead of image streams or image stream tags, is that you can control all the parameters of the Kubernetes plug-in pod template.

Sample config map for jenkins-agent

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: jenkins-agent
  labels:
    role: jenkins-agent
data:
  template1: |-
    <org.csanchez.jenkins.plugins.kubernetes.PodTemplate>
    <inheritFrom></inheritFrom>
    <name>template1</name>
    <instanceCap>2147483647</instanceCap>
    <idleMinutes>0</idleMinutes>
    <label>template1</label>
    <serviceAccount>jenkins</serviceAccount>
    <nodeSelector></nodeSelector>
    <volumes/>
    <containers>
    <org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
    <name>jnlp</name>
    <image>openshift/jenkins-agent-maven-35-centos7:v3.10</image>
    <privileged>>false</privileged>
    <alwaysPullImage>>true</alwaysPullImage>
    <workingDir>/tmp</workingDir>
    <command></command>
    <args>${computer.jnlpMac} ${computer.name}</args>
    <ttyEnabled>>false</ttyEnabled>
    <resourceRequestCpu></resourceRequestCpu>
    <resourceRequestMemory></resourceRequestMemory>
    <resourceLimitCpu></resourceLimitCpu>
    <resourceLimitMemory></resourceLimitMemory>
    <envVars/>
    </org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
    </containers>
    <envVars/>
    <annotations/>
```

```
<imagePullSecrets/>
<nodeProperties/>
</org.csanchez.jenkins.plugins.kubernetes.PodTemplate>
```



NOTE

If you log in to the Jenkins console and make further changes to the pod template configuration after the pod template is created, and the OpenShift Container Platform Sync plug-in detects that the config map has changed, it will replace the pod template and overwrite those configuration changes. You cannot merge a new configuration with the existing configuration.

Do not log in to the Jenkins console and modify the pod template configuration. If you do so after the pod template is created, and the OpenShift Container Platform Sync plug-in detects that the image associated with the image stream or image stream tag has changed, it replaces the pod template and overwrites those configuration changes. You cannot merge a new configuration with the existing configuration.

Consider the config map approach if you have more complex configuration needs.

After it is installed, the OpenShift Container Platform Sync plug-in monitors the API server of OpenShift Container Platform for updates to image streams, image stream tags, and config maps and adjusts the configuration of the Kubernetes plug-in.

The following rules apply:

- Removing the label or annotation from the config map, image stream, or image stream tag results in the deletion of any existing **PodTemplate** from the configuration of the Kubernetes plug-in.
- If those objects are removed, the corresponding configuration is removed from the Kubernetes plug-in.
- Either creating appropriately labeled or annotated **ConfigMap**, **ImageStream**, or **ImageStreamTag** objects, or the adding of labels after their initial creation, leads to creating of a **PodTemplate** in the Kubernetes-plugin configuration.
- In the case of the **PodTemplate** by config map form, changes to the config map data for the **PodTemplate** are applied to the **PodTemplate** settings in the Kubernetes plug-in configuration and overrides any changes that were made to the **PodTemplate** through the Jenkins UI between changes to the config map.

To use a container image as a Jenkins agent, the image must run the agent as an entrypoint. For more details about this, refer to the official [Jenkins documentation](#).

12.2.7. Jenkins permissions

If in the config map the **<serviceAccount>** element of the pod template XML is the OpenShift Container Platform service account used for the resulting pod, the service account credentials are mounted into the pod. The permissions are associated with the service account and control which operations against the OpenShift Container Platform master are allowed from the pod.

Consider the following scenario with service accounts used for the pod, which is launched by the Kubernetes Plug-in that runs in the OpenShift Container Platform Jenkins image.

If you use the example template for Jenkins that is provided by OpenShift Container Platform, the **jenkins** service account is defined with the **edit** role for the project Jenkins runs in, and the master Jenkins pod has that service account mounted.

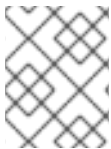
The two default Maven and NodeJS pod templates that are injected into the Jenkins configuration are also set to use the same service account as the Jenkins master.

- Any pod templates that are automatically discovered by the OpenShift Container Platform sync plug-in because their image streams or image stream tags have the required label or annotations are configured to use the Jenkins master service account as their service account.
- For the other ways you can provide a pod template definition into Jenkins and the Kubernetes plug-in, you have to explicitly specify the service account to use. Those other ways include the Jenkins console, the **podTemplate** pipeline DSL that is provided by the Kubernetes plug-in, or labeling a config map whose data is the XML configuration for a pod template.
- If you do not specify a value for the service account, the **default** service account is used.
- Ensure that whatever service account is used has the necessary permissions, roles, and so on defined within OpenShift Container Platform to manipulate whatever projects you choose to manipulate from the within the pod.

12.2.8. Creating a Jenkins service from a template

Templates provide parameter fields to define all the environment variables with predefined default values. OpenShift Container Platform provides templates to make creating a new Jenkins service easy. The Jenkins templates should be registered in the default **openshift** project by your cluster administrator during the initial cluster setup.

The two available templates both define deployment configuration and a service. The templates differ in their storage strategy, which affects whether or not the Jenkins content persists across a pod restart.



NOTE

A pod might be restarted when it is moved to another node or when an update of the deployment configuration triggers a redeployment.

- **jenkins-ephemeral** uses ephemeral storage. On pod restart, all data is lost. This template is only useful for development or testing.
- **jenkins-persistent** uses a Persistent Volume (PV) store. Data survives a pod restart.

To use a PV store, the cluster administrator must define a PV pool in the OpenShift Container Platform deployment.

After you select which template you want, you must instantiate the template to be able to use Jenkins.

Procedure

1. Create a new Jenkins application using one of the following methods:

- A PV:

```
$ oc new-app jenkins-persistent
```

- Or an **emptyDir** type volume where configuration does not persist across pod restarts:

```
$ oc new-app jenkins-ephemeral
```

12.2.9. Using the Jenkins Kubernetes plug-in

In the following example, the **openshift-jee-sample BuildConfig** object causes a Jenkins Maven agent pod to be dynamically provisioned. The pod clones some Java source code, builds a WAR file, and causes a second **BuildConfig**, **openshift-jee-sample-docker** to run. The second **BuildConfig** layers the new WAR file into a container image.

Sample BuildConfig that uses the Jenkins Kubernetes plug-in

```
kind: List
apiVersion: v1
items:
- kind: ImageStream
  apiVersion: v1
  metadata:
    name: openshift-jee-sample
- kind: BuildConfig
  apiVersion: v1
  metadata:
    name: openshift-jee-sample-docker
  spec:
    strategy:
      type: Docker
    source:
      type: Docker
      dockerfile: |-
        FROM openshift/wildfly-101-centos7:latest
        COPY ROOT.war /wildfly/standalone/deployments/ROOT.war
        CMD $STI_SCRIPTS_PATH/run
      binary:
        asFile: ROOT.war
    output:
      to:
        kind: ImageStreamTag
        name: openshift-jee-sample:latest
- kind: BuildConfig
  apiVersion: v1
  metadata:
    name: openshift-jee-sample
  spec:
    strategy:
      type: JenkinsPipeline
      jenkinsPipelineStrategy:
        jenkinsfile: |-
          node("maven") {
            sh "git clone https://github.com/openshift/openshift-jee-sample.git ."
            sh "mvn -B -Popenshift package"
            sh "oc start-build -F openshift-jee-sample-docker --from-file=target/ROOT.war"
          }
    triggers:
      - type: ConfigChange
```


It is also possible to override the specification of the dynamically created Jenkins agent pod. The following is a modification to the previous example, which overrides the container memory and specifies an environment variable.

Sample BuildConfig that the Jenkins Kubernetes Plug-in, specifying memory limit and environment variable

```
kind: BuildConfig
apiVersion: v1
metadata:
  name: openshift-jee-sample
spec:
  strategy:
    type: JenkinsPipeline
    jenkinsPipelineStrategy:
      jenkinsfile: |-
        podTemplate(label: "mypod", 1
          cloud: "openshift", 2
          inheritFrom: "maven", 3
          containers: [
            containerTemplate(name: "jnlp", 4
              image: "openshift/jenkins-agent-maven-35-centos7:v3.10", 5
              resourceRequestMemory: "512Mi", 6
              resourceLimitMemory: "512Mi", 7
              envVars: [
                envVar(key: "CONTAINER_HEAP_PERCENT", value: "0.25") 8
              ]
            )
          ]
        ) {
          node("mypod") { 9
            sh "git clone https://github.com/openshift/openshift-jee-sample.git ."
            sh "mvn -B -Popenshift package"
            sh "oc start-build -F openshift-jee-sample-docker --from-file=target/ROOT.war"
          }
        }
      }
    triggers:
      - type: ConfigChange
```

- 1 A new pod template called **mypod** is defined dynamically. The new pod template name is referenced in the node stanza.
- 2 The **cloud** value must be set to **openshift**.
- 3 The new pod template can inherit its configuration from an existing pod template. In this case, inherited from the Maven pod template that is pre-defined by OpenShift Container Platform.
- 4 This example overrides values in the pre-existing container, and must be specified by name. All Jenkins agent images shipped with OpenShift Container Platform use the Container name **jnlp**.
- 5 Specify the container image name again. This is a known issue.
- 6 A memory request of **512 Mi** is specified.
- 7 A memory limit of **512 Mi** is specified.

- 8 An environment variable **CONTAINER_HEAP_PERCENT**, with value **0.25**, is specified.
- 9 The node stanza references the name of the defined pod template.

By default, the pod is deleted when the build completes. This behavior can be modified with the plug-in or within a pipeline Jenkinsfile.

12.2.10. Jenkins memory requirements

When deployed by the provided Jenkins Ephemeral or Jenkins Persistent templates, the default memory limit is **1 Gi**.

By default, all other process that run in the Jenkins container cannot use more than a total of **512 MiB** of memory. If they require more memory, the container halts. It is therefore highly recommended that pipelines run external commands in an agent container wherever possible.

And if **Project** quotas allow for it, see recommendations from the Jenkins documentation on what a Jenkins master should have from a memory perspective. Those recommendations proscribe to allocate even more memory for the Jenkins master.

It is recommended to specify memory request and limit values on agent containers created by the Jenkins Kubernetes plug-in. Admin users can set default values on a per-agent image basis through the Jenkins configuration. The memory request and limit parameters can also be overridden on a per-container basis.

You can increase the amount of memory available to Jenkins by overriding the **MEMORY_LIMIT** parameter when instantiating the Jenkins Ephemeral or Jenkins Persistent template.

12.2.11. Additional Resources

- See [Base image options](#) for more information on the [Red Hat Universal Base Images \(UBI\)](#).

12.3. JENKINS AGENT

OpenShift Container Platform provides three images that are suitable for use as Jenkins agents: the **Base**, **Maven**, and **Node.js** images.

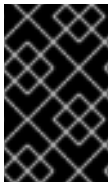
The first is a base image for Jenkins agents:

- It pulls in both the required tools, headless Java, the Jenkins JNLP client, and the useful ones including **git**, **tar**, **zip**, and **nss** among others.
- It establishes the JNLP agent as the endpoint.
- It includes the **oc** client tooling for invoking command line operations from within Jenkins jobs.
- It provides Dockerfiles for both Red Hat Enterprise Linux (RHEL) and **localdev** images.

Two more images that extend the base image are also provided:

- Maven v3.5 image
- Node.js v10 image and Node.js v12 image

The Maven and Node.js Jenkins agent images provide Dockerfiles for the Universal Base Image (UBI) that you can reference when building new agent images. Also note the **contrib** and **contrib/bin** subdirectories. They allow for the insertion of configuration files and executable scripts for your image.



IMPORTANT

Use and extend an appropriate agent image version for the your of OpenShift Container Platform. If the **oc** client version that is embedded in the agent image is not compatible with the OpenShift Container Platform version, unexpected behavior can result.

12.3.1. Jenkins agent images

The OpenShift Container Platform Jenkins agent images are available on [Quay.io](https://quay.io) or registry.redhat.io.

Jenkins images are available through the Red Hat Registry:

```
$ docker pull registry.redhat.io/openshift4/ose-jenkins:<v4.5.0>
```

```
$ docker pull registry.redhat.io/openshift4/jenkins-agent-nodejs-10-rhel7:<v4.5.0>
```

```
$ docker pull registry.redhat.io/openshift4/jenkins-agent-nodejs-12-rhel7:<v4.5.0>
```

```
$ docker pull registry.redhat.io/openshift4/ose-jenkins-agent-maven:<v4.5.0>
```

```
$ docker pull registry.redhat.io/openshift4/ose-jenkins-agent-base:<v4.5.0>
```

To use these images, you can either access them directly from [Quay.io](https://quay.io) or registry.redhat.io or push them into your OpenShift Container Platform container image registry.

12.3.2. Jenkins agent environment variables

Each Jenkins agent container can be configured with the following environment variables.

Variable	Definition	Example values and settings
----------	------------	-----------------------------

Variable	Definition	Example values and settings
JAVA_MAX_HEAP_PARAM, CONTAINER_HEAP_PERCENT, JENKINS_MAX_HEAP_UPPER_BOUND_MB	<p>These values control the maximum heap size of the Jenkins JVM. If JAVA_MAX_HEAP_PARAM is set, its value takes precedence. Otherwise, the maximum heap size is dynamically calculated as CONTAINER_HEAP_PERCENT of the container memory limit, optionally capped at JENKINS_MAX_HEAP_UPPER_BOUND_MB MiB.</p> <p>By default, the maximum heap size of the Jenkins JVM is set to 50% of the container memory limit with no cap.</p>	<p>JAVA_MAX_HEAP_PARAM example setting: -Xmx512m</p> <p>CONTAINER_HEAP_PERCENT default: 0.5, or 50%</p> <p>JENKINS_MAX_HEAP_UPPER_BOUND_MB example setting: 512 MiB</p>
JAVA_INITIAL_HEAP_PARAM, CONTAINER_INITIAL_PERCENT	<p>These values control the initial heap size of the Jenkins JVM. If JAVA_INITIAL_HEAP_PARAM is set, its value takes precedence. Otherwise, the initial heap size is dynamically calculated as CONTAINER_INITIAL_PERCENT of the dynamically calculated maximum heap size.</p> <p>By default, the JVM sets the initial heap size.</p>	<p>JAVA_INITIAL_HEAP_PARAM example setting: -Xms32m</p> <p>CONTAINER_INITIAL_PERCENT example setting: 0.1, or 10%</p>
CONTAINER_CORE_LIMIT	<p>If set, specifies an integer number of cores used for sizing numbers of internal JVM threads.</p>	<p>Example setting: 2</p>
JAVA_TOOL_OPTIONS	<p>Specifies options to apply to all JVMs running in this container. It is not recommended to override this value.</p>	<p>Default: - XX:+UnlockExperimentalVMOptions - XX:+UseCGroupMemoryLimitForHeap - Dsun.zip.disableMemoryMapping=true</p>
JAVA_GC_OPTS	<p>Specifies Jenkins JVM garbage collection parameters. It is not recommended to override this value.</p>	<p>Default: -XX:+UseParallelGC - XX:MinHeapFreeRatio=5 - XX:MaxHeapFreeRatio=10 - XX:GCTimeRatio=4 - XX:AdaptiveSizePolicyWeight=90</p>

Variable	Definition	Example values and settings
JENKINS_JAVA_OVERRIDES	Specifies additional options for the Jenkins JVM. These options are appended to all other options, including the Java options above, and can be used to override any of them, if necessary. Separate each additional option with a space and if any option contains space characters, escape them with a backslash.	Example settings: -Dfoo -Dbar; -Dfoo=first\ value -Dbar=second\ value

12.3.3. Jenkins agent memory requirements

A JVM is used in all Jenkins agents to host the Jenkins JNLP agent as well as to run any Java applications such as **javac**, Maven, or Gradle.

By default, the Jenkins JNLP agent JVM uses 50% of the container memory limit for its heap. This value can be modified by the **CONTAINER_HEAP_PERCENT** environment variable. It can also be capped at an upper limit or overridden entirely.

By default, any other processes run in the Jenkins agent container, such as shell scripts or **oc** commands run from pipelines, cannot use more than the remaining 50% memory limit without provoking an OOM kill.

By default, each further JVM process that runs in a Jenkins agent container uses up to 25% of the container memory limit for its heap. It might be necessary to tune this limit for many build workloads.

12.3.4. Jenkins agent Gradle builds

Hosting Gradle builds in the Jenkins agent on OpenShift Container Platform presents additional complications because in addition to the Jenkins JNLP agent and Gradle JVMs, Gradle spawns a third JVM to run tests if they are specified.

The following settings are suggested as a starting point for running Gradle builds in a memory constrained Jenkins agent on OpenShift Container Platform. You can modify these settings as required.

- Ensure the long-lived Gradle daemon is disabled by adding **org.gradle.daemon=false** to the **gradle.properties** file.
- Disable parallel build execution by ensuring **org.gradle.parallel=true** is not set in the **gradle.properties** file and that **--parallel** is not set as a command line argument.
- To prevent Java compilations running out-of-process, set **java { options.fork = false }** in the **build.gradle** file.
- Disable multiple additional test processes by ensuring **test { maxParallelForks = 1 }** is set in the **build.gradle** file.
- Override the Gradle JVM memory parameters by the **GRADLE_OPTS**, **JAVA_OPTS** or **JAVA_TOOL_OPTIONS** environment variables.

- Set the maximum heap size and JVM arguments for any Gradle test JVM by defining the **maxHeapSize** and **jvmArgs** settings in **build.gradle**, or through the **-Dorg.gradle.jvmargs** command line argument.

12.3.5. Jenkins agent pod retention

Jenkins agent pods, are deleted by default after the build completes or is stopped. This behavior can be changed by the Kubernetes plug-in pod retention setting. Pod retention can be set for all Jenkins builds, with overrides for each pod template. The following behaviors are supported:

- **Always** keeps the build pod regardless of build result.
- **Default** uses the plug-in value, which is the pod template only.
- **Never** always deletes the pod.
- **On Failure** keeps the pod if it fails during the build.

You can override pod retention in the pipeline Jenkinsfile:

```
podTemplate(label: "mypod",
  cloud: "openshift",
  inheritFrom: "maven",
  podRetention: onFailure(), 1
  containers: [
    ...
  ]) {
  node("mypod") {
    ...
  }
}
```

- 1** Allowed values for **podRetention** are **never()**, **onFailure()**, **always()**, and **default()**.



WARNING

Pods that are kept might continue to run and count against resource quotas.

12.4. SOURCE-TO-IMAGE

You can use the [Red Hat Software Collections](#) images as a foundation for applications that rely on specific runtime environments such as Node.js, Perl, or Python. Special versions of some of these runtime base images are referred to as Source-to-Image (S2I) images. With S2I images, you can insert your code into a base image environment that is ready to run that code.

S2I images include:

- Java
- Node.js

- Perl
- PHP
- Python
- Ruby

S2I images are available for you to use directly from the OpenShift Container Platform web UI by selecting **Catalog** → **Developer Catalog**.

S2I images are also available through the [Configuring the Cluster Samples Operator](#).

12.4.1. Source-to-image build process overview

Source-to-image (S2I) produces ready-to-run images by injecting source code into a container that prepares that source code to be run. It performs the following steps:

1. Runs the **FROM <builder image>** command
2. Copies the source code to a defined location in the builder image
3. Runs the assemble script in the builder image
4. Sets the run script in the builder image as the default command

Buildah then creates the container image.

12.4.2. Additional resources

- For instructions on using the Cluster Samples Operator, see the [Configuring the Cluster Samples Operator](#).
- For more information on S2I builds, see the [builds strategy documentation on S2I builds](#).
- For troubleshooting assistance for the S2I process, see [Troubleshooting the Source-to-Image process](#).
- For an overview of creating images with S2I, see [Creating images from source code with source-to-image](#).
- For an overview of testing S2I images, see [About testing S2I images](#).

12.5. CUSTOMIZING SOURCE-TO-IMAGE IMAGES

Source-to-image (S2I) builder images include assemble and run scripts, but the default behavior of those scripts is not suitable for all users. You can customize the behavior of an S2I builder that includes default scripts.

12.5.1. Invoking scripts embedded in an image

Builder images provide their own version of the source-to-image (S2I) scripts that cover the most common use-cases. If these scripts do not fulfill your needs, S2I provides a way of overriding them by adding custom ones in the **.s2i/bin** directory. However, by doing this, you are completely replacing the standard scripts. In some cases, replacing the scripts is acceptable, but, in other scenarios, you can run a

few commands before or after the scripts while retaining the logic of the script provided in the image. To reuse the standard scripts, you can create a wrapper script that runs custom logic and delegates further work to the default scripts in the image.

Procedure

1. Look at the value of the **io.openshift.s2i.scripts-url** label to determine the location of the scripts inside of the builder image:

```
$ podman inspect --format='{{ index .Config.Labels "io.openshift.s2i.scripts-url" }}'
wildfly/wildfly-centos7
```

Example output

```
image:///usr/libexec/s2i
```

You inspected the **wildfly/wildfly-centos7** builder image and found out that the scripts are in the **/usr/libexec/s2i** directory.

2. Create a script that includes an invocation of one of the standard scripts wrapped in other commands:

.s2i/bin/assemble script

```
#!/bin/bash
echo "Before assembling"

/usr/libexec/s2i/assemble
rc=$?

if [ $rc -eq 0 ]; then
    echo "After successful assembling"
else
    echo "After failed assembling"
fi

exit $rc
```

This example shows a custom assemble script that prints the message, runs the standard assemble script from the image, and prints another message depending on the exit code of the assemble script.



IMPORTANT

When wrapping the run script, you must use **exec** for invoking it to ensure signals are handled properly. The use of **exec** also precludes the ability to run additional commands after invoking the default image run script.

.s2i/bin/run script

```
#!/bin/bash
echo "Before running application"
exec /usr/libexec/s2i/run
```