# OpenShift Container Platform 4.18

## Images

Creating and managing images and imagestreams in OpenShift Container Platform

Last Updated: 2026-04-01

# OpenShift Container Platform 4.18 Images

Creating and managing images and imagestreams in OpenShift Container Platform

## Legal Notice



## 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. OVERVIEW OF IMAGES

To understand how containerized applications work in OpenShift Container Platform, you need to know about containers, images, and image streams. This overview explains these core concepts and how they work together in your cluster.

## 1.1. IMAGES

Container images are binaries that include all requirements for running a single container. You can use images to package applications and deploy them consistently across multiple containers and hosts in OpenShift Container Platform.

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**.

**Additional resources**

- Creating images
- Managing images
- Using images

## 1.2. IMAGE REGISTRY

An image registry is a content server that stores and serves container images in OpenShift Container Platform. You can use registries to access container images from external sources or the integrated registry in OpenShift Container Platform.

Registries contain 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 OpenShift image registry for managing custom container images.

## 1.3. IMAGE REPOSITORY

An image repository is a collection of related container images and tags that identify them. You can use image repositories to organize and manage related container images in OpenShift Container Platform.

For example, the OpenShift Container Platform Jenkins images are in the following repository:

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

## 1.4. UNDERSTANDING IMAGE TAGS IN IMAGE STREAMS

Image tags in OpenShift Container Platform help you organize, identify, and reference specific versions of container images in image streams. Tags are human-readable labels that act as pointers to particular image layers and digests.

Tags function as mutable pointers within an image stream. When a new image is imported or tagged into the stream, the tag is updated to point to the new image's immutable SHA digest. A single image digest can have multiple tags simultaneously assigned to it. For example, the **:v3.11.59-2** and **:latest** tags are assigned to the same image digest.

Tags offer two main benefits:

- Tags serve as the primary mechanism for builds and deployments to request a specific version of an image from an image stream.
- Tags help maintain clarity and allow for easy promotion of images between environments. For example, you can promote an image from the **:test** tag to the **:prod** tag.

While image tags are primarily used for referencing images in configurations, OpenShift Container Platform provides the **oc tag** command for managing tags directly within image streams. This command is similar to the **podman tag** or **docker tag** commands, but it operates on image streams instead of directly on local images. It is used to create a new tag pointer or update an existing tag pointer within an image stream to point to a new image.

Image tags are appended to the image name or image stream name by using a colon (**:**) as a separator.

| Context | Syntax Format | Example |
|---|---|---|
| External Registry | **<registry_path>:<tag>** | **registry.access.redhat.com/openshift3/jenkins-2-rhel7:v3.11.59-2** |
| Local Image Stream | **<image_stream_name>:<tag>** | **jenkins:latest** |

## 1.5. IMAGE IDS

Image IDs are Secure Hash Algorithm (SHA) codes that uniquely identify container images in OpenShift Container Platform. You can use image IDs to pull specific versions of images that never change.

For example, the following image ID is for the **docker.io/openshift/jenkins-2-centos7** image:

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

## 1.6. CONTAINERS

Containers are isolated running instances of container images that serve as the basic units of OpenShift Container Platform applications. By understanding containers, you can work with containerized applications and manage how they run in your cluster.

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. By using the **podman** CLI, you can experiment with containers separately from OpenShift Container Platform.

## 1.7. USING IMAGE STREAMS

Image streams provide an abstraction for referencing container images from within OpenShift Container Platform. You can use image streams to manage image versions and automate builds and deployments in your cluster.

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 does 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.

**Additional resources**

- Managing image streams
- Using image streams with Kubernetes resources
- Triggering updates on image stream updates

## 1.8. IMAGE STREAM TAGS

Image stream tags are named pointers to images in image streams in OpenShift Container Platform. You can configure image stream tags to reference specific versions of container images.

## 1.9. IMAGE STREAM IMAGES

Image stream images are API resource objects in OpenShift Container Platform that retrieve specific container images from image streams. You can use image stream images to access metadata about particular image SHA identifiers.

## 1.10. IMAGE STREAM TRIGGERS

Image stream triggers in OpenShift Container Platform cause specific actions when image stream tags change. You can configure triggers to automatically start builds or deployments when new images are imported.

For example, importing a new image 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.

## 1.11. HOW YOU CAN USE THE CLUSTER SAMPLES OPERATOR

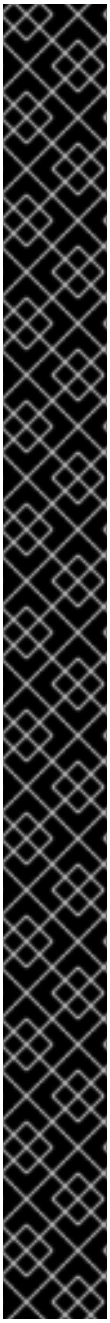
To manage sample image streams and templates in OpenShift Container Platform, you can use the Cluster Samples Operator. The Cluster Samples Operator creates default samples during initial startup to initiate image streams and templates in the **openshift** namespace.

## 1.12. ADDITIONAL RESOURCES

- Configuring the Cluster Samples Operator
- Use the Operator with an alternate registry
- Understanding templates
- Creating applications using Ruby on Rails

# CHAPTER 2. CONFIGURING THE CLUSTER SAMPLES OPERATOR

The Cluster Samples Operator in Red Hat Enterprise Linux (RHEL) installs and updates image streams and templates in the **openshift** namespace. Configure the Operator to manage sample content and customize which image streams and templates are available in your cluster.



**IMPORTANT**

- The Cluster Samples Operator is deprecated. No new templates, samples, or non-Source-to-Image (Non-S2I) image streams are added to the Cluster Samples Operator. However, the existing S2I builder image streams and templates will continue to receive updates until the Cluster Samples Operator is removed in a future release. S2I image streams and templates include:
  - Ruby
  - Python
  - Node.js
  - Perl
  - PHP
  - HTTPD
  - Nginx
  - EAP
  - Java
  - Webserver
  - .NET
  - Go
- The Cluster Samples Operator will stop managing and providing support to the non-S2I samples (image streams and templates). You can contact the image stream or template owner for any requirements and future plans. In addition, refer to the following link:
  - List of the repositories hosting the image stream or templates

## 2.1. UNDERSTANDING THE CLUSTER SAMPLES OPERATOR

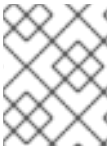
During installation, the Operator creates the default configuration object for itself and then creates the sample image streams and templates, including quick start templates.

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. The deployment of the Operator is within the **openshift-cluster-samples-operator** namespace.

The image for the Cluster Samples Operator has 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 by using the default configuration.

If the Cluster Samples Operator is removed during installation, you can use the Cluster Samples Operator with an alternate registry so that content can be imported. Then you can set the Cluster Samples Operator to **Managed** to get the samples. Use the following instructions:

- Using the Cluster Samples Operator with an alternate registry

For more information about configuring credentials, see the following link:

- Using image pull secrets

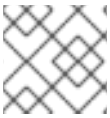
## 2.2. CLUSTER SAMPLES OPERATOR USE OF MANAGEMENT STATE

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 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 the registry after three minutes on initial startup after a clean installation.
- If the Cluster Samples Operator detects that it is on an IPv6 network.
- If the image controller configuration parameters prevent the creation of image streams by using the default image registry, or by using the image registry specified by **samplesRegistry** setting. For more information, see the following links:
  - Image controller configuration parameters
  - Cluster Samples Operator configuration parameters



NOTE

For OpenShift Container Platform, the default image registry is **registry.redhat.io**.

However, if the Cluster Samples Operator detects that it is on an IPv6 network and an OpenShift Container Platform global proxy is configured, then the IPv6 check supersedes all the checks. As a result, the Cluster Samples Operator bootstraps itself as **Removed**.

IMPORTANT

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

### 2.2.1. Restricted network installation

The Cluster Samples Operator boostrapping itself as **Removed** when unable to access **registry.redhat.io** facilitates restricted network installations when the network restriction is already in place.

As a cluster administrator, you have more time to decide if samples are needed when the Operator is boostrapped **Removed**. This is because the Cluster Samples Operator does not submit alerts that sample image stream imports are failing when the management state is **Removed**. When the Cluster Samples Operator management state is **Managed**, and the Operator attempts to install sample image streams, failing-import alerts start two hours after initial installation.

### 2.2.2. Restricted network installation with initial network access

If a cluster that eventually runs on a restricted network is first installed while network access exists, the Cluster Samples Operator installs content from **registry.redhat.io**.

In this case, you can defer samples installation until you have decided which samples are needed by overriding the default configuration of **Managed** for a connected installation.

If you want the Cluster Samples Operator to bootstrap with the management state as **Removed** during an installation that has initial network access, override the Cluster Samples Operator default configuration by using the following instructions:

- Customizing nodes

To host samples in your restricted environment, use the following instructions:

- Using the Cluster Samples Operator with an alternate registry

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

**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
```

## 2.3. CLUSTER SAMPLES OPERATOR 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 at a rate of about every 15 minutes until either one of the following occurs:

- The import succeeds.
- The Cluster Samples Operator configuration is changed such that either the image stream is added to the **skippedImagestreams** list, or the management state is changed to **Removed**.

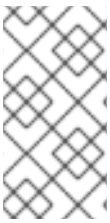
### 2.3.1. Cluster Samples Operator assistance for mirroring

During installation, OpenShift Container Platform creates a config map named **imagestreamtag-to-image** in the **openshift-cluster-samples-operator** namespace.

The **imagestreamtag-to-image** config map contains an entry, the populating image, for each image stream tag.

The format of the key for each entry in the data field in the config map is **<image_stream_name>_<image_stream_tag_name>**.

During a disconnected installation of OpenShift Container Platform, the status of the Cluster Samples Operator is set to **Removed**. If you choose to change it to **Managed**, it installs samples.



NOTE

The use of samples in a network-restricted or discontinued environment might require access to services external to your network. Some example services include: Github, Maven Central, npm, RubyGems, PyPi and others. There might be additional steps to take that allow the Cluster Samples Operators objects to reach the services they require.

Use the following principles to determine which images you need to mirror for your image streams to import:

- While the Cluster Samples Operator is set to **Removed**, you can create your mirrored registry, or determine which existing mirrored registry you want to use.
- Mirror the samples you want to the mirrored registry using the new config map as your guide.
- Add any of the image streams you did not mirror to the **skippedImagestreams** list of the Cluster Samples Operator configuration object.
- Set **samplesRegistry** of the Cluster Samples Operator configuration object to the mirrored registry.
- Then set the Cluster Samples Operator to **Managed** to install the image streams you have mirrored.

## 2.4. CLUSTER SAMPLES OPERATOR CONFIGURATION PARAMETERS

The samples resource offers the following configuration fields:

| Parameter | Description |
|---|---|
| **managementState** | **Managed**: The Cluster Samples Operator updates the samples as the configuration dictates.<br><br>**Unmanaged**: The Cluster Samples Operator ignores updates to its configuration resource object and any image streams or templates in the **openshift** namespace.<br><br>**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. |
| **samplesRegistry** | 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.<br><br>  <br><br>NOTE<br><br>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.<br><br>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. |

| Parameter | Description |
|---|---|
| **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.

### 2.4.1. Configuration restrictions

When the Cluster Samples Operator starts supporting multiple architectures, you cannot change the architecture list while the Operator is in the **Managed** state.

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 ensure a clean slate. Removing all samples before switching is not required.

### 2.4.2. Samples resource conditions

The samples resource maintains the following conditions in its status:

| Condition | Description |
|---|---|
| **SamplesExists** | Indicates the samples are created in the **openshift** namespace. |
| **ImageChangesInProgress** | **True** when image streams are created or updated, but not all of the tag spec generations and tag status generations match.<br><br>**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.<br><br>This condition is deprecated in OpenShift Container Platform. |

| Condition | Description |
| --- | --- |
| **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** | Indicator of which image streams had errors during the image import phase for one of their tags.<br><br>**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. |
| **MigrationInProgress** | **True** when the Cluster Samples Operator detects that the version is different from the Cluster Samples Operator version with which the current samples set are installed.<br><br>This condition is deprecated in OpenShift Container Platform. |

## 2.5. ACCESSING THE CLUSTER SAMPLES OPERATOR CONFIGURATION

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

### Prerequisites

- You installed the OpenShift CLI (**oc**).

### Procedure

- Access the Cluster Samples Operator configuration by running the following command:

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

The Cluster Samples Operator configuration resembles the following example:

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

## 2.6. REMOVING DEPRECATED IMAGE STREAM TAGS FROM THE CLUSTER SAMPLES OPERATOR

The Cluster Samples Operator leaves deprecated image stream tags in an image stream because users can have deployments that use the deprecated image stream tags.

You can remove deprecated image stream tags by editing the image stream with the **oc tag** command.

NOTE

Deprecated image stream tags that the samples providers have removed from their image streams are not included on initial installations.

**Prerequisites**

- You installed the OpenShift CLI (**oc**).

**Procedure**

- Remove deprecated image stream tags by editing the image stream with the following **oc tag** command:

```
$ oc tag -d <image_stream_name:tag>
```

**Example output**

```
Deleted tag default/<image_stream_name:tag>.
```

# CHAPTER 3. USING THE CLUSTER SAMPLES OPERATOR WITH AN ALTERNATE REGISTRY

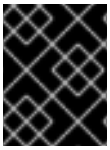
You can use the Cluster Samples Operator with an alternate registry by first creating a mirror registry. Before you create the mirror registry, you must prepare the mirror host.

## 3.1. ABOUT THE MIRROR REGISTRY

You must have access to the internet to obtain the necessary container images. Using an alternative registry means that you place the mirror registry on a mirror host that has access to both your network and the internet.

You can mirror the images that are required for OpenShift Container Platform installation and subsequent product updates to a container mirror registry such as Red Hat Quay, JFrog Artifactory, Sonatype Nexus Repository, or Harbor. If you do not have access to a large-scale container registry, you can use the *mirror registry for Red Hat OpenShift*, a small-scale container registry included with OpenShift Container Platform subscriptions.

You can use any container registry that supports Docker v2-2, such as Red Hat Quay, the *mirror registry for Red Hat OpenShift*, Artifactory, Sonatype Nexus Repository, or Harbor. Regardless of your chosen registry, the procedure to mirror content from Red Hat hosted sites on the internet to an isolated image registry is the same. After you mirror the content, you configure each cluster to retrieve this content from your mirror registry.



**IMPORTANT**

The OpenShift image registry cannot be used as the target registry because it does not support pushing without a tag, which is required during the mirroring process.

If choosing a container registry that is not the *mirror registry for Red Hat OpenShift*, it 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 your 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*.

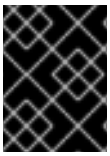
For mirrored registries, to view the source of pulled images, you must review the **Trying to access** log entry in the CRI-O logs. Other methods to view the image pull source, such as using the **crictl images** command on a node, show the non-mirrored image name, even though the image is pulled from the mirrored location.

**NOTE**

Red Hat does not test third party registries with OpenShift Container Platform.

### 3.1.1. Installing the OpenShift CLI on Linux

To manage your cluster and deploy applications from the command line, install the OpenShift CLI (**oc**) binary on Linux.



**IMPORTANT**

If you installed an earlier version of **oc**, you cannot use it to complete all of the commands in OpenShift Container Platform 4.18. Download and install the new version of **oc**.

**Procedure**

1. Navigate to the OpenShift Container Platform downloads page on the Red Hat Customer Portal.
2. Select the architecture from the **Product Variant** drop-down list.
3. Select the appropriate version from the **Version** drop-down list.
4. Click **Download Now** next to the **OpenShift v4.18 Linux Clients** entry and save the file.
5. Unpack the archive:

```
$ tar xvf <file>
```

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

```
$ echo $PATH
```

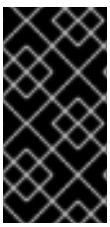
**Verification**

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

```
$ oc <command>
```

### 3.1.2. Installing the OpenShift CLI on Windows

To manage your cluster and deploy applications from the command line, install OpenShift CLI (**oc**) binary on Windows.



**IMPORTANT**

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

Download and install the new version of **oc**.

**Procedure**

1. Navigate to the Download OpenShift Container Platform page on the Red Hat Customer Portal.
2. Select the appropriate version from the **Version** list.
3. Click **Download Now** next to the **OpenShift v4.18 Windows Client** entry and save the file.

4. Extract the archive with a ZIP program.
5. Move the **oc** binary to a directory that is on your **PATH** variable.
   To check your **PATH** variable, open the command prompt and execute the following command:

```
C:\> path
```

**Verification**

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

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

### 3.1.3. Installing the OpenShift CLI on macOS

To manage your cluster and deploy applications from the command line, install the OpenShift CLI (**oc**) binary on macOS.

**IMPORTANT**

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

Download and install the new version of **oc**.

**Procedure**

1. Navigate to the Download OpenShift Container Platform page on the Red Hat Customer Portal.
2. Select the architecture from the **Product Variant** list.
3. Select the appropriate version from the **Version** list.
4. Click **Download Now** next to the **OpenShift v4.18 macOS Clients** entry and save the file.

**NOTE**

For macOS arm64, choose the **OpenShift v4.18 macOS arm64 Client** entry.

5. Unpack and unzip the archive.
6. Move the **oc** binary to a directory on your **PATH** variable.
   To check your **PATH** variable, open a terminal and execute the following command:

```
$ echo $PATH
```

**Verification**

- Verify your installation by using an **oc** command:

```
$ oc <command>
```

## 3.2. CONFIGURING CREDENTIALS THAT ALLOW IMAGES TO BE MIRRORED

Create a container image registry credentials file so that you can mirror images from Red Hat to your mirror. Complete the following steps on the installation host.

### Prerequisites

- You configured a mirror registry to use in your disconnected environment.

### Procedure

1. Download your **registry.redhat.io** pull secret from Red Hat OpenShift Cluster Manager .
2. Make a copy of your pull secret in JSON format by running the following command:

```
$ cat ./pull-secret | jq . > <path>/<pull_secret_file_in_json>
```

   Specify the path to the directory to store the pull secret in and a name for the JSON file that you create.

   **Example pull secret**

```
{
  "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"
    }
  }
}
```

3. Generate the base64-encoded user name and password or token for your mirror registry by running the following command:

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

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

   **Example output**

```
BGVtbYk3ZHAtqXs=
```

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

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

- For the **<mirror_registry>** value, 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:8443**.
- For the **<credentials>** value, specify the base64-encoded user name and password for the mirror registry.

**Example modified pull secret**

```
{
  "auths": {
    "registry.example.com": {
      "auth": "BGVtbYk3ZHAtqXs=",
      "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"
    }
  }
}
```

## 3.3. MIRRORING THE OPENSHIFT CONTAINER PLATFORM IMAGE REPOSITORY

Mirror the OpenShift Container Platform image repository to your registry to use during cluster installation or upgrade. Complete the following steps on the mirror host.

**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 Red Hat OpenShift Cluster Manager and modified it to include authentication to your mirror repository.
- If you use self-signed certificates, you have specified a Subject Alternative Name in the certificates.

**Procedure**

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 following 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 **4.20.1**.

   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 cluster:

```
$ ARCHITECTURE=<cluster_architecture>
```

Specify the architecture of the cluster, such as **x86_64**, **aarch64**, **s390x**, or **ppc64le**.

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

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

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

3. Mirror the version images to the mirror 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}
```

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

**IMPORTANT**

Running the **oc image mirror** command might result in the following error: **error: unable to retrieve source image**. This error occurs when image indexes include references to images that no longer exist on the image registry. Image indexes might retain older references to allow users running those images an upgrade path to newer points on the upgrade graph. As a temporary workaround, you can use the **--skip-missing** option to bypass the error and continue downloading the image index. For more information, see Service Mesh Operator mirroring failed.

- 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:

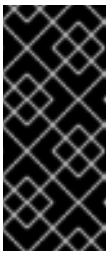
   - If your mirror host does not have internet access, run the following command:

   ```
   $ oc adm release extract -a ${LOCAL_SECRET_JSON} --icsp-file=<file> --
   command=openshift-install
   "${LOCAL_REGISTRY}/${LOCAL_REPOSITORY}:${OCP_RELEASE}-
   ${ARCHITECTURE}" \
   --insecure=true
   ```

   Optional: If you do not want to configure trust for the target registry, add the **--insecure=true** flag.

   - 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.

5. For clusters using installer-provisioned infrastructure, run the following command:

```
$ openshift-install
```

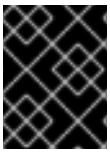
## 3.4. USING CLUSTER SAMPLES OPERATOR IMAGE STREAMS WITH ALTERNATE OR MIRRORED REGISTRIES

You can use an alternate or mirror registry to host your images streams instead of using the Red Hat registry.

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.

**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.



**IMPORTANT**

The Cluster Samples Operator must be set to **Managed** in a disconnected environment. To install the image streams, you must have a mirrored registry.

**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/rhscl/ruby-25-rhel7:latest ${MIRROR_ADDR}/rhscl/ruby-
```

```
25-rhel7:latest
```

3. Create the image configuration object for the cluster by running the following command:

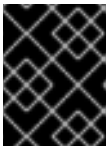
```
$ oc create configmap registry-config --from-
file=${MIRROR_ADDR_HOSTNAME}..5000=$path/ca.crt -n openshift-config
```

4. Add the required trusted CAs for the mirror in the 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
```



IMPORTANT

This step 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

The Cluster Samples Operator issues alerts if image stream imports are failing but the Cluster Samples Operator is either periodically retrying or does not appear to be retrying them.

Many of the templates in the **openshift** namespace reference the image streams. You can use **Removed** to purge both the image streams and templates. This eliminates the possibility of attempts to use the templates if they are not functional because of any missing image streams.

### 3.4.1. Cluster Samples Operator assistance for mirroring

During installation, OpenShift Container Platform creates a config map named **imagestreamtag-to-image** in the **openshift-cluster-samples-operator** namespace.

The **imagestreamtag-to-image** config map contains an entry, the populating image, for each image stream tag.

The format of the key for each entry in the data field in the config map is **<image_stream_name>_<image_stream_tag_name>**.

During a disconnected installation of OpenShift Container Platform, the status of the Cluster Samples Operator is set to **Removed**. If you choose to change it to **Managed**, it installs samples.

NOTE

The use of samples in a network-restricted or discontinued environment might require access to services external to your network. Some example services include: Github, Maven Central, npm, RubyGems, PyPi and others. There might be additional steps to take that allow the Cluster Samples Operators objects to reach the services they require.

Use the following principles to determine which images you need to mirror for your image streams to import:

- While the Cluster Samples Operator is set to **Removed**, you can create your mirrored registry, or determine which existing mirrored registry you want to use.
- Mirror the samples you want to the mirrored registry using the new config map as your guide.
- Add any of the image streams you did not mirror to the **skippedImagestreams** list of the Cluster Samples Operator configuration object.
- Set **samplesRegistry** of the Cluster Samples Operator configuration object to the mirrored registry.
- Then set the Cluster Samples Operator to **Managed** to install the image streams you have mirrored.

## 3.5. ADDITIONAL RESOURCES

- Viewing the image pull source
- Using Cluster Samples Operator image streams with alternate or mirrored registries

# 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 OpenShift image 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

Follow fundamental guidelines for creating container images to ensure they are secure, efficient, and reproducible for deployment in OpenShift Container Platform.

#### 4.1.1.1. 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.

#### 4.1.1.2. 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 **image** and it currently includes version **1.0**, you might provide a tag of **image: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 **image: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 **image: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 **image:latest** takes on the risk of any incompatible changes being introduced.

#### 4.1.1.3. 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.

#### 4.1.1.4. 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.

#### 4.1.1.5. 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.

#### 4.1.1.6. 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.

#### 4.1.1.7. 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.

#### 4.1.1.8. 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**.

#### 4.1.1.9. 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.

#### 4.1.1.10. 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.

#### 4.1.1.11. 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

Use the integrated image-building capabilities of OpenShift Container Platform to create, manage, and deploy reproducible container images directly from source code or Dockerfiles.

#### 4.1.2.1. 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.

#### 4.1.2.2. 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 the sensitive areas of a container. If applied to sensitive areas, such as the **/etc/passwd** file, such changes can allow the modification of these files by unintended users, potentially exposing the container or host. CRI-O supports the insertion of arbitrary user IDs into a container's **/etc/passwd** file. As such, changing permissions is never required.

Additionally, the **/etc/passwd** file should not exist in any container image. If it does, the CRI-O container runtime will fail to inject a random UID into the **/etc/passwd** file. In such cases, the container might face challenges in resolving the active UID. Failing to meet this requirement could impact the functionality of certain containerized applications.

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 **anyuid** security context constraint (SCC). Alternatively, you can allow all images to run as any user.

### 4.1.2.3. 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.

### 4.1.2.4. 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.

#### 4.1.2.5. 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.

NOTE

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.

#### 4.1.2.6. 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.

#### 4.1.2.7. 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.

#### 4.1.2.8. 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.

#### 4.1.2.9. 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.

#### 4.1.2.10. 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

Define comprehensive image metadata during creation to ensure OpenShift Container Platform correctly configures image runtime settings and tracks image lineage and compliance. This helps to provide 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** | 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.<br>`LABEL io.openshift.tags mongodb,mongodb24,nosql` |
| **io.openshift.wants** | 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.<br>`LABEL io.openshift.wants mongodb,redis` |
| **io.k8s.description** | 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.<br>`LABEL io.k8s.description The MySQL 5.5 Server with master-slave replication support` |
| **io.openshift.non-scalable** | 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**.<br>`LABEL io.openshift.non-scalable true` |
| **io.openshift.min-memory** and **io.openshift.min-cpu** | 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.<br>`LABEL io.openshift.min-memory 16Gi`<br>`LABEL io.openshift.min-cpu 4` |

## 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

Leverage the Source-to-image (S2I) process in OpenShift Container Platform to seamlessly transform application source code into ready-to-run, reproducible container images.

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

- Sources
- 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

Define mandatory Source-to-image (S2I) scripts, such as **assemble** and **run** to enable OpenShift Container Platform to build, customize, and execute highly reproducible container images.

You can write 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** | 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:<br>1. Optional: Restore build artifacts. If you want to support incremental builds, make sure to define **save-artifacts** as well.<br>2. Place the application source in the desired location.<br>3. Build the application artifacts.<br>4. Install the artifacts into locations appropriate for them to run. |
| **run** | The **run** script executes your application. This script is required. |
| **save-artifacts** | The **save-artifacts** script gathers all dependencies that can speed up the build processes that follow. This script is optional. For example:<br>- For Ruby, **gems** installed by Bundler.<br>- For Java, **.m2** contents.<br>These dependencies are gathered into a **tar** file and streamed to the standard output. |
| **usage** | The **usage** script allows you to inform the user how to properly use your image. This script is optional. |
| **test/run** | 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:<br>1. Build the image.<br>2. Run the image to verify the **usage** script.<br>3. Run **s2i build** to verify the **assemble** script.<br>4. Optional: Run **s2i build** again to verify the **save-artifacts** and **assemble** scripts save and restore artifacts functionality.<br>5. Run the image to verify the test application is working.<br>  <br>NOTE<br>The suggested location to put the test application built by your **test/run** script is the **test/test-app** directory in your image repository. |

### 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

Verify your Source-to-image (S2I) environment and resulting application images to ensure successful, reproducible container deployment within OpenShift Container Platform.

As an 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 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 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

Image streams in OpenShift Container Platform provide a layer of abstraction over container images, enabling automation for your CI/CD pipelines. You can configure builds and deployments to watch image streams and automatically trigger new builds or deployments when images are updated.

The main advantage of using image streams is the automation they enable for your continuous integration and continuous delivery (CI/CD) pipelines. For example:

- Image streams allow OpenShift Container Platform resources like Builds and Deployments to "watch" them.
- When a new image is added to the stream, or when an existing tag is modified to point to a new image, the watching resources receive notifications.
- When notifications are received, the watching resources can automatically react by performing a new build or a new deployment.

## 5.2. TAGGING IMAGES

Image tags identify specific versions of container images in image streams. You can use image tags to organize images and control which versions your builds and deployments use.

### 5.2.1. Understanding image tags in image streams

Image tags in OpenShift Container Platform help you organize, identify, and reference specific versions of container images in image streams. Tags are human-readable labels that act as pointers to particular image layers and digests.

Tags function as mutable pointers within an image stream. When a new image is imported or tagged into the stream, the tag is updated to point to the new image's immutable SHA digest. A single image digest can have multiple tags simultaneously assigned to it. For example, the **:v3.11.59-2** and **:latest** tags are assigned to the same image digest.

Tags offer two main benefits:

- Tags serve as the primary mechanism for builds and deployments to request a specific version of an image from an image stream.
- Tags help maintain clarity and allow for easy promotion of images between environments. For example, you can promote an image from the **:test** tag to the **:prod** tag.

While image tags are primarily used for referencing images in configurations, OpenShift Container Platform provides the **oc tag** command for managing tags directly within image streams. This command is similar to the **podman tag** or **docker tag** commands, but it operates on image streams instead of directly on local images. It is used to create a new tag pointer or update an existing tag pointer within an image stream to point to a new image.

Image tags are appended to the image name or image stream name by using a colon (**:**) as a separator.

| Context | Syntax Format | Example |
| --- | --- | --- |
| External Registry | **<registry_path>:<tag>** | **registry.access.redhat.com/openshift3/jenkins-2-rhel7:v3.11.59-2** |
| Local Image Stream | **<image_stream_name>:<tag>** | **jenkins:latest** |

### 5.2.2. Image tag conventions

Image tag naming conventions in OpenShift Container Platform provide guidelines for creating tags that enable effective image pruning and maintain manageable image streams. Use consistent naming patterns to avoid tags that point to single revisions and never update.

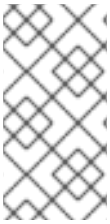
Tags that are too specific effectively pin the tag to a single image revision that is never updated. For example, if you create a tag named **v2.0.1-may-2019**, the tag points to just one revision of an image and is never updated. If you use default image pruning options, such an image is never removed.

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.

To ensure proper garbage collection, use broader, more generic tags that are designed to be updated when a new image revision is built. The following table provides some recommended tagging conventions using the format **<image_name>:<image_tag>**.

**Table 5.1. Image tag naming conventions**

| Description | Example |
| --- | --- |
| **Major/Minor Version** (Ideal for mutable pointers) | **myimage:v2.0** |
| **Full Revision** (Often used for tracking, but requires manual pruning) | **myimage:v2.0.1** |
| Architecture | **myimage:v2.0-x86_64** |
| Base image | **myimage:v1.2-centos7** |
| Latest | **myimage:latest** |
| Latest stable | **myimage:stable** |



NOTE

If your team requires the use of unique, date-specific, or highly revisioned tags like **v2.0.1-may-2019**, you must 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

To organize images and create aliases for specific versions or automatically track changes to source tags in OpenShift Container Platform, you can add tags to image streams with the **oc tag** command.

There are two types of tags available in OpenShift Container Platform:

- Permanent tags: A permanent tag 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.
- Tracking tags: A tracking tag means that the destination tag's metadata is updated during the import of the source tag.

The default behavior creates a permanent tag that is pinned to an image ID.

**Procedure**

- Optional: Add a tag to an image stream by entering the following command. The default behavior creates a permanent tag that is pinned to an image ID:

```
$ oc tag <source_reference> <destination_image_stream>:<destination_tag>
```

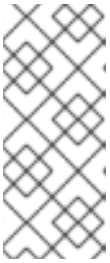
For example, to configure the **ruby** image stream **static-2.0** tag to always refer to the specific image that the **ruby:2.0** tag points to now, enter the following command:

```
$ 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.

- Optional: Use the **--alias=true** flag to create a tracking tag. This ensures the destination tag automatically updates (tracks) when the source tag changes to point to a new image. For example, to ensure that the **ruby:latest** tag always reflects whatever image is currently tagged as **ruby:2.0**, enter the following command:

```
$ oc tag --alias=true ruby:2.0 ruby:latest
```



NOTE

A Tracking Tag created with **--alias=true** automatically updates its image ID whenever the source tag changes. Use the **latest** or **stable** tracking tags for creating common, long-lived aliases. This tracking behavior only works correctly within a single image stream. Trying to create a cross-image stream alias produces an error.

- Optional: Use the **--scheduled=true** flag to have the destination tag be refreshed, or re-imported, periodically. The period is configured globally at the system level. For example:

```
$ oc tag <source_reference> <destination_image_stream>:<destination_tag> --scheduled=true
```

- Optional: Use the **--reference** flag to create an image stream tag that is not imported. The tag permanently points to the source location, regardless of changes to the source image. For example:

```
$ oc tag <source_reference> <destination_image_stream>:<destination_tag> --reference
```

- Optional. Use the **--insecure** flag if the source registry is not secured with a valid HTTPS certificate. This flag tells the image stream to skip certificate verification during the import progress. For example:

```
$ oc tag <source_reference> <destination_image_stream>:<destination_tag> --insecure
```

- Optional: Use the **--reference-policy=local** flag to instruct OpenShift Container Platform to always fetch the tagged image from the integrated registry. 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 **--reference-policy=local** flag also allows for pulling from insecure registries without a need to supply the **--insecure** flag to the container runtime provided that the image stream has an insecure annotation or the tag has an insecure import policy. For example:

```
$ oc tag <source_reference> <destination_image_stream>:<destination_tag> --reference-policy=local
```

### 5.2.4. Removing tags from image streams

To keep your image streams clean and maintain organized image references in OpenShift Container Platform, you can remove unused or outdated image stream tags. Remove tags by using the **oc delete istag** or **oc tag -d** commands.

**Procedure**

- Remove a tag from an image stream by entering the following command:

```
$ oc delete istag/<name>:<tag>
```

  For example, to remove the **ruby:latest** tag from the **ruby** image stream, enter the following command:

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

- Alternatively, you can remove a tag using the **oc tag -d** command:

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

  For example, to remove the **ruby:latest** tag from the **ruby** image stream, enter the following command:

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

### 5.2.5. Using image stream reference syntax

To ensure that your builds and deployments use the intended image version in OpenShift Container Platform, you must use the correct reference syntax format.

**Procedure**

- To reference an image by a mutable tag (**ImageStreamTag**) from an image stream within your cluster, use the **<image_stream_name>:<tag>** format in your build or deployment. For example:

```
# ...
spec:
  containers:
    - name: my-app
      image: <image_stream_name>:<tag>
```

where:

**spec.containers.image**

Specifies the image to use from the image stream. For example, **ruby:2.0**.

- To reference a specific, immutable image ID, or digest, within an image stream, use the **<image_stream_name>@<image_id>** format in your build or deployment. For example:

```
# ...
spec:
  containers:
    - name: my-app
      image: <image_stream_name>@<image_id>
```

where:

**spec.containers.image**

Specifies the image to use from the image stream. For example, **ruby@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b28e**.

NOTE

Using the image ID with the **@id** syntax ensures your configuration always uses the exact same image, even if the tag is later updated to point to a different image.

- To reference an image from an external registry by using the DockerImage format, use the standard Docker pull specification: **<registry>/<namespace>/<image_name>:<tag>**. For example:

```
# ...
spec:
```

```
source:
  type: Dockerfile
strategy:
  type: Docker
  dockerStrategy:
    from:
      kind: DockerImage
      name: <registry>/<namespace>/<image_name>:<tag>
```

where:

**spec.strategy.dockerStrategy.from.name**

Specifies the image to use from the external registry. For example, **registry.redhat.io/rhel7:latest**.

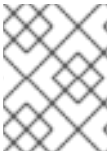
NOTE

When no tag is specified in a **DockerImage** reference, the **latest** tag is assumed.

### 5.2.6. Understanding image stream reference types

By using image streams in OpenShift Container Platform, you can reference container images by using different reference types. These reference types define which specific image version your builds and deployments use.

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



NOTE

Example image stream definitions often contain definitions of **ImageStreamTag** and references to **DockerImage**, but never contain definitions of **ImageStreamImage**.

Table 5.2. Imagestream reference types

| Reference Type | Description | Syntax Examples |
|---|---|---|
| **ImageStreamTag** | References or retrieves an image for a given image stream and human-readable tag. | **image_stream_name :tag** |
| **ImageStreamImage** | References or retrieves an image for a given image stream and immutable SHA ID (digest). | **image_stream_name @id** |
| **DockerImage** | References or retrieves an image from an external registry. Uses the standard **docker pull** specification. | **openshift/ruby-20-centos7:2.0**, **registry.redhat.io/rhel7:latest**, **centos/ruby-22-centos7@sha256:3a335d7d…** |

## 5.3. IMAGE PULL POLICY

To manage image updates and optimize pod startup performance in OpenShift Container Platform, you can configure the **imagePullPolicy** parameter in your container specifications. This setting controls when container images are pulled from registries.

### 5.3.1. About the imagePullPolicy parameter

To control when OpenShift Container Platform pulls container images from registries or uses locally cached copies when starting containers, you can configure the **imagePullPolicy** parameter. This policy helps you manage image updates and optimize pod startup performance.

The following table lists the possible values for the **imagePullPolicy** parameter:

**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. |
| **Never** | Never pull the image. |

The following example sets the **imagePullPolicy** parameter to **IfNotPresent** for the image tagged **v1.2.3**:

**Example imagePullPolicy configuration**

```
apiVersion: apps/v1
kind: Deployment
# ...
spec:
  # ...
  template:
    spec:
      containers:
      - name: my-app-container
        image: registry.example.com/myapp:v1.2.3
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
```

where:

**spec.template.spec.containers.image**

Specifies is the image to use. In this example, the image tag is explicitly set to **v1.2.3**.

**spec.template.spec.containers.imagePullPolicy**

Specifies the policy to use. In this example, the policy is set to **IfNotPresent** because the image tag is not **latest**.

#### 5.3.1.1. Omitting the imagePullPolicy parameter

When you omit the **imagePullPolicy** parameter, OpenShift Container Platform automatically determines the policy based on the image tag. This default behavior ensures that the **latest** tag always pulls the newest image, while specific version tags use locally cached images when available to improve efficiency.

| Image tag | **imagePullPolicy** setting | Behavior |
|---|---|---|
| **latest** | **Always** | Always pulls the image. This policy helps ensure that the container always uses the latest version of the image. |
| Any other tag (for example, **v1.2.3**, **stable**, **production**) | **IfNotPresent** | Pull only if necessary. This policy uses the locally cached version of the image if it exists on the node, avoiding unnecessary pulls from the registry. |

## 5.4. USING IMAGE PULL SECRETS

To authenticate with container registries and pull images across OpenShift Container Platform projects or from secured registries, you can configure and use image pull secrets.

You first obtain the registry authentication credentials, which are typically found in the **~/.docker/config.json** file for Docker or the **~/.config/containers/auth.json** file for Podman, created by the pull secret from Red Hat OpenShift Cluster Manager process. This content is then used to create or update the global **pullSecret** object within your cluster, allowing access to images from quay.io and registry.redhat.io.

NOTE

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

### 5.4.1. Allowing pods to reference images across projects

To allow pods in one OpenShift Container Platform project to reference images from another project, you can bind a service account to the **system:image-puller** role in the target project. Use the **oc policy add-role-to-user** or **oc policy add-role-to-group** command to grant cross-project image access.

NOTE

When you create a pod service account or a namespace, wait until the service account is provisioned with a Docker pull secret. If you create a pod before its service account is fully provisioned, the pod fails to access the OpenShift image registry.

**Procedure**

1. Allow pods in **project-a** to reference images in **project-b** by entering the following command. In this example, the service account **default** in **project-a** is bound 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
```

2. Optional: Allow access for any service account in **project-a** by using the **add-role-to-group** flag. For example:

```
$ 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

Pull secrets enable pods in OpenShift Container Platform to authenticate with secured registries and pull container images. Docker and Podman store authentication credentials in configuration files that you can use to create pull secrets for your service accounts.

The following files store your authentication information if you have previously logged in to a secured or insecure registry:

- **Docker**: By default, Docker uses **$HOME/.docker/config.json**.
- **Podman**: By default, Podman uses **$HOME/.config/containers/auth.json**.

NOTE

Both Docker and Podman credential files and the associated pull secret can contain multiple references to the same registry if they have unique paths, for example, **quay.io** and **quay.io/<example_repository>**. However, neither Docker nor Podman support multiple entries for the exact same registry path.

**Example config.json file**

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

#### Example pull secret

```
apiVersion: v1
data:
  .dockerconfigjson:
ewogICAiYXV0aHMiOnsKICAgICAgIm0iOnsKICAgICAgIsKICAgICAgICAgImF1dGgiOiJiM0JsYj0iLAogI
CAgICAgICAiZW1haWwiOiJ5b3VAZXhhbXBsZS5jb20iCiAgICAgIH0KICAgfQp9Cg==
kind: Secret
metadata:
  creationTimestamp: "2021-09-09T19:10:11Z"
  name: pull-secret
  namespace: default
  resourceVersion: "37676"
  uid: e2851531-01bc-48ba-878c-de96cfe31020
type: Opaque
```

### 5.4.2.1. Creating a pull secret

To authenticate with container registries in OpenShift Container Platform, you can create pull secrets from existing Docker or Podman authentication files. You can also create secrets by providing registry credentials directly by using the **oc create secret docker-registry** command.

#### Procedure

1. Create a secret from an existing authentication file:

   a. For Docker clients using **.docker/config.json**, enter the following command:

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

   b. For Podman clients using **.config/containers/auth.json**, enter the following command:

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

2. Optional: If you do not already have a Docker credentials file for the secured registry, you can create a secret by running the following command:

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

### 5.4.2.2. Using a pull secret in a workload

To allow workloads to pull images from private registries in OpenShift Container Platform, you can link the pull secret to a service account by entering the **oc secrets link** command or by defining it directly in your workload configuration YAML file.

**Procedure**

1. Link the pull secret to a service account by entering the following command. Note that the name of the service account should match the name of the service account that pod uses. The default service account is **default**.

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

2. Verify the change by entering the following command:

```
$ oc get serviceaccount default -o yaml
```

**Example output**

```
apiVersion: v1
imagePullSecrets:
- name: default-dockercfg-123456
- name: <pull_secret_name>
kind: ServiceAccount
metadata:
  annotations:
    openshift.io/internal-registry-pull-secret-ref: <internal_registry_pull_secret>
  creationTimestamp: "2025-03-03T20:07:52Z"
  name: default
  namespace: default
  resourceVersion: "13914"
  uid: 9f62dd88-110d-4879-9e27-1ffe269poe3
secrets:
- name: <pull_secret_name>
```

3. Optional: Instead of linking the secret to a service account, you can alternatively reference it directly in your pod or workload definition. This is useful for GitOps workflows such as ArgoCD. For example:

**Example pod specification**

```
apiVersion: v1
kind: Pod
metadata:
  name: <secure_pod_name>
spec:
  containers:
  - name: <container_name>
    image: quay.io/my-private-image
  imagePullSecrets:
  - name: <pull_secret_name>
```

**Example ArgoCD workflow**

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: <example_workflow>
spec:
```

```
entrypoint: <main_task>
imagePullSecrets:
- name: <pull_secret_name>
```

#### 5.4.2.3. Pulling from private registries with delegated authentication

To pull images from private registries that delegate authentication to a separate service in OpenShift Container Platform, you can create pull secrets for both the authentication server and the registry endpoint. Use the **oc create secret docker-registry** command to create separate secrets for each service.

**Procedure**

1. Create a secret for the delegated authentication server by entering the following command:

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

2. Create a secret for the private registry by entering the following command:

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

### 5.4.3. Updating the global cluster pull secret

To add new registries or change authentication for your OpenShift Container Platform cluster, you can update the global pull secret by replacing it or appending new credentials. Use the **oc set data secret/pull-secret** command to apply the updated pull secret to all nodes in your cluster.

**IMPORTANT**

To transfer your cluster to another owner, you must initiate the transfer in OpenShift Cluster Manager and then update the pull secret on the cluster. Updating a cluster's pull secret without initiating the transfer in OpenShift Cluster Manager causes the cluster to stop reporting Telemetry metrics in OpenShift Cluster Manager.

For more information, see "Transferring cluster ownership" in the Red Hat OpenShift Cluster Manager documentation.

**Prerequisites**

- You have access to the cluster as a user with the **cluster-admin** role.

**Procedure**

1. Optional: To append a new pull secret to the existing pull secret:

   a. Download the pull secret by entering the following command:

   ```
   $ oc get secret/pull-secret -n openshift-config --template='{{index .data
   ".dockerconfigjson" | base64decode}}' > <pull_secret_location>
   ```

   where:

   **<pull_secret_location>**

   Specifies the path to the pull secret file.

   b. Add the new pull secret by entering the following command:

   ```
   $ oc registry login --registry="<registry>" \
   --auth-basic="<username>:<password>" \
   --to=<pull_secret_location>
   ```

   where:

   **<registry>**

   Specifies the new registry. You can include many repositories within the same registry, for example: **--registry="<registry/my-namespace/my-repository>**.

   **<username>:<password>**

   Specifies the credentials of the new registry.

   **<pull_secret_location>**

   Specifies the path to the pull secret file.

2. Update the global pull secret for your cluster by entering the following command. Note that this update rolls out to all nodes, which can take some time depending on the size of your cluster.

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

   where:

   **<pull_secret_location>**

   Specifies the path to the new pull secret file.

### Additional resources

- Transferring cluster ownership

# CHAPTER 6. MANAGING IMAGE STREAMS

To create and update container images and track version changes in OpenShift Container Platform, you can use image streams and tags. Add, update, remove, and import image stream tags to manage your container images.

## 6.1. USING IMAGE STREAMS

Image streams provide an abstraction for referencing container images from within OpenShift Container Platform. You can use image streams to manage image versions and automate builds and deployments in your cluster.

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 does 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

To customize image retrieval and security policies for your applications, configure image streams within OpenShift Container Platform. This process lets you define image pull specifications, manage tags, and control access permissions necessary for reliable application deployment.

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

### Imagestream object definition

```
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  annotations:
    openshift.io/generated-by: OpenShiftNewApp
  labels:
    app: ruby-sample-build
    template: application-template-stibuild
  name: origin-ruby-sample
  namespace: test
spec: {}
status:
  dockerImageRepository: 172.30.56.218:5000/test/origin-ruby-sample
  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: 2017-09-01T13:40:11Z
      dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab3fc5
      generation: 1
      image: sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d
    tag: latest
```

where

**name**

Specifies the name of the image stream

**ruby-sample**

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

**dockerImageReference**

Specifies the SHA identifier that this image stream tag currently references. Resources that reference this image stream tag use this identifier

**image**

Specifies the SHA identifier that this image stream tag previously referenced. You can use it to rollback to an older image.

**tag**

Specifies the image stream tag name.

## 6.3. IMAGE STREAM IMAGES

To precisely identify and manage the actual image content associated with a specific tag, reference and use image stream images in OpenShift Container Platform. This ensures your application deployments reliably target immutable image definitions.

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

To maintain human-readable references to immutable images, utilize image stream tags within OpenShift Container Platform. These tags are essential because they enable your builds and deployments to accurately target specific, stable image content.

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**

```
kind: ImageStream
apiVersion: image.openshift.io/v1
metadata:
  name: my-image-stream
# ...
  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: 2017-09-01T13:40:11Z
      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

To automate your application lifecycle and ensure they use the latest code, configure image stream triggers in OpenShift Container Platform. 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

Manage how OpenShift Container Platform tracks newly uploaded images by understanding image stream mapping. When the integrated registry receives a new image, it automatically creates and sends an image stream mapping, providing the image's crucial project, name, tag, and 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: image.openshift.io/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
      - OPENSHIFT_BUILD_NAMESPACE=test
      - OPENSHIFT_BUILD_SOURCE=https://github.com/openshift/ruby-hello-world.git
      - EXAMPLE=sample-app
      - OPENSHIFT_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
      - OPENSHIFT_BUILD_NAME=ruby-sample-build-1
      - OPENSHIFT_BUILD_NAMESPACE=test
      - OPENSHIFT_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

To organize and manage container images in OpenShift Container Platform, you can use image streams and image stream tags. By using image streams, you can track image versions and simplify deployments.

**IMPORTANT**

Do not run workloads in or share access to default projects. Default projects are reserved for running core cluster components.

The following default projects are considered highly privileged: **default**, **kube-public**, **kube-system**, **openshift**, **openshift-infra**, **openshift-node**, and other system-created projects that have the **openshift.io/run-level** label set to **0** or **1**. Functionality that relies on admission plugins, such as pod security admission, security context constraints, cluster resource quotas, and image reference resolution, does not work in highly privileged projects.

### 6.7.1. Getting information about image streams

To efficiently manage and monitor your image streams in OpenShift Container Platform, retrieve information about their versions. You can get general information about the image stream and detailed information about all the tags it is pointing to, ensuring your deployed applications rely on the correct image versions.

**Procedure**

- To get general information about the image stream and detailed information about all the tags it is pointing to, enter the following command:

```
$ 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
```

- To get all of the information available about a particular image stream tag, enter the following command:

```
$ 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.

- Enter the following command to discover which architecture or operating system that an image stream tag supports:

```
$ oc get istag <image-stream-tag> -ojsonpath="{range .image.dockerImageManifests[*]}
{.os}/{.architecture}{'\n'}{end}"
```

For example:

```
$ oc get istag busybox:latest -ojsonpath="{range .image.dockerImageManifests[*]}
{.os}/{.architecture}{'\n'}{end}"
```

**Example output**

```
linux/amd64
linux/arm
linux/arm64
linux/386
linux/mips64le
linux/ppc64le
linux/riscv64
linux/s390x
```

## 6.7.2. Adding tags to an image stream

To accurately manage and track specific versions of your container images, add tags to your image streams within OpenShift Container Platform, This ensures reliable referencing and deployment throughout your environment.

**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

To enable OpenShift Container Platform resources to track and consume container images sourced from external registries, add tags to the corresponding image streams. This action integrates external image content securely into your cluster's local image management system.

**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

To maintain flexibility and consistency in deployment definitions, update an image stream tag to reflect a different tag in OpenShift Container Platform. Specifically, you can update a tag to reflect another tag in an image stream, which is essential for managing image versions effectively.

#### 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@sha256:438208801c4806548460b27bd1fbcb7bb188273d13871ab43f.
```

### 6.7.5. Removing image stream tags

To maintain control over your image history and simplify management within OpenShift Container Platform, you can remove old tags from an image stream. This action helps ensure that your resources track only the current and necessary image references.

#### Procedure

- Remove old tags from an image stream:

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

For example:

```
$ oc tag -d python:3.6
```

#### Example output

```
Deleted tag default/python:3.6
```

#### Additional resources

- Removing deprecated image stream tags from the Cluster Samples Operator

### 6.7.6. Configuring periodic importing of image stream tags

To maintain up-to-date image definitions from an external container image registry, configure periodic importing of image stream tags. This process allows you to quickly re-import images for critical security updates by using 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 will reset its behavior to default.

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

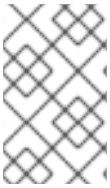
## 6.8. IMPORTING AND WORKING WITH IMAGES AND IMAGE STREAMS

To bring container images into your OpenShift Container Platform cluster and manage their references, you can import images from external registries and organize them by using image streams. By using this process, you can maintain a centralized registry of container images for your applications.

The following sections describe how to import, and work with, image streams.

### 6.8.1. Importing images and image streams from private registries

To securely manage content from external sources, configure your image streams to import tag and image metadata from private registries requiring authentication. This procedure is essential if you change the registry that the Cluster Samples Operator uses for pulling content to something other than the default 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.

### 6.8.2. Working with manifest lists

To precisely manage multi-architecture or variant images contained within a manifest list, use the **--import-mode** flag with **oc import-image** or **oc tag** CLI commands. This functionality allows you to import a single sub-manifest, or all manifests, of a manifest list, providing fine-grained control over your image stream content.

In some cases, users might want to use sub-manifests directly. When **oc adm prune images** is run, or the **CronJob** pruner runs, they cannot detect when a sub-manifest list is used. As a result, an administrator using **oc adm prune images**, or the **CronJob** pruner, might delete entire manifest lists, including sub-manifests.

To avoid this limitation, you can use the manifest list by tag or by digest instead.

**Procedure**

- Create an image stream that includes multi-architecture images, and sets the import mode to **PreserveOriginal**, by entering the following command:

```
$ oc import-image <multiarch-image-stream-tag>  --from=
<registry>/<project_name>/<image-name> \
--import-mode='PreserveOriginal' --reference-policy=local --confirm
```

**Example output**

```
---
Arch:           <none>
Manifests:      linux/amd64
sha256:6e325b86566fafd3c4683a05a219c30c421fbccbf8d87ab9d20d4ec1131c3451
                linux/arm64
sha256:d8fad562ffa75b96212c4a6dc81faf327d67714ed85475bf642729703a2b5bf6
                linux/ppc64le
sha256:7b7e25338e40d8bdeb1b28e37fef5e64f0afd412530b257f5b02b30851f416e1
---
```

- Alternatively, enter the following command to import an image with the **Legacy** import mode, which discards manifest lists and imports a single sub-manifest:

```
$ oc import-image <multiarch-image-stream-tag>  --from=
<registry>/<project_name>/<image-name> \
--import-mode='Legacy' --confirm
```

NOTE

The **--import-mode=** default value is **Legacy**. Excluding this value, or failing to specify either **Legacy** or **PreserveOriginal**, imports a single sub-manifest. An invalid import mode returns the following error: **error: valid ImportMode values are Legacy or PreserveOriginal**.

#### 6.8.2.1. Configuring periodic importing of manifest lists

To maintain up-to-date image references for complex, multi-architecture images, configure periodic importing of manifest lists. To periodically re-import a manifest list, you can use the **--scheduled** flag, ensuring your image stream tracks the latest versions from external registries.

**Procedure**

- Set the image stream to periodically update the manifest list by entering the following command:

```
$ oc import-image <multiarch-image-stream-tag>  --from=
<registry>/<project_name>/<image-name> \
--import-mode='PreserveOriginal' --scheduled=true
```

#### 6.8.2.2. Configuring SSL/TLS when importing manifest lists

To control connection security and access policies for manifest lists sourced from external repositories, configure SSL/TLS settings during image importing. To configure SSL/TLS when importing a manifest list, you can use the **--insecure** flag to bypass standard certificate validation requirements if necessary.

**Procedure**

- Set **--insecure=true** so that importing a manifest list skips SSL/TLS verification. For example:

```
$ oc import-image <multiarch-image-stream-tag> --from=<registry>/<project_name>/<image-
name> \
--import-mode='PreserveOriginal' --insecure=true
```

### 6.8.3. Specifying architecture for --import-mode

To control the architecture of your imported images and ensure proper deployment, use the **--import-mode=** flag. You can swap your imported image stream between multi-architecture and single architecture by excluding or including the **--import-mode=** flag as needed.

**Procedure**

- Run the following command to update your image stream from multi-architecture to single architecture by excluding the **--import-mode=** flag:

```
$ oc import-image <multiarch-image-stream-tag> --from=<registry>/<project_name>/<image-name>
```

- Run the following command to update your image stream from single-architecture to multi-architecture:

```
$ oc import-image <multiarch_image_stream_tag>  --from=
<registry>/<project_name>/<image_name> \
--import-mode='PreserveOriginal'
```

### 6.8.4. Configuration fields for --import-mode

To implement multi-architecture image management using the **--import-mode** flag, reference the necessary configuration fields. These fields define precise parameters for selecting and importing specific manifests into your OpenShift Container Platform cluster.

The following table describes the options available for the **--import-mode=** flag:

| Parameter | Description |
|---|---|
| **Legacy** | The default option for **--import-mode**. When specified, the manifest list is discarded, and a single sub-manifest is imported. The platform is chosen in the following order of priority:<br>1. Tag annotations<br>2. Control plane architecture<br>3. Linux/AMD64<br>4. The first manifest in the list |
| **PreserveOriginal** | When specified, the original manifest is preserved. For manifest lists, the manifest list and all of its sub-manifests are imported. |

# CHAPTER 7. USING IMAGE STREAMS WITH KUBERNETES RESOURCES

To use image streams with both OpenShift Container Platform native resources and standard Kubernetes resources, reference them in your resource definitions. Image streams work with resources such as **Build**, **DeploymentConfigs**, **Job**, **ReplicationController**, **ReplicaSet**, and **Deployment** resources.

## 7.1. ENABLING IMAGE STREAMS WITH KUBERNETES RESOURCES

When using Kubernetes resources, you must reference image streams located within the same project by specifying a single segment value, such as **ruby:2.5**, which identifies the image stream name and its tag. This ensures the resource correctly targets the local image stream within its scope.

IMPORTANT

Do not run workloads in or share access to default projects. Default projects are reserved for running core cluster components.

The following default projects are considered highly privileged: **default**, **kube-public**, **kube-system**, **openshift**, **openshift-infra**, **openshift-node**, and other system-created projects that have the **openshift.io/run-level** label set to **0** or **1**. Functionality that relies on admission plugins, such as pod security admission, security context constraints, cluster resource quotas, and image reference resolution, does not work in highly privileged projects.

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.

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

**Procedure**

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: image.openshift.io/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
```

3. Optional: 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
```

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

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

# CHAPTER 8. TRIGGERING UPDATES ON IMAGE STREAM CHANGES

When image stream tags update in OpenShift Container Platform, the platform automatically rolls out new images to deployments and builds that reference those tags. You configure this automatic triggering behavior differently depending on the type of resource that uses the image stream.

## 8.1. OPENSHIFT 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

To enable Kubernetes resources, such as **Deployments** and **StatefulSets**, to seamlessly consume new image versions, configure image stream change triggers in OpenShift Container Platform. This ensures your application deployments are automatically updated when the associated image stream detects a change.

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:

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    image.openshift.io/triggers:
      [
       {
         "from": {
           "kind": "ImageStreamTag",
           "name": "example:latest",
           "namespace": "myapp"
         },
         "fieldPath": "spec.template.spec.containers[?(@.name==\"web\")].image",
         "paused": false
       },
      # ...
      ]
# ...
```

where:

**kind**

Specifies the resource to trigger from, and must have the value **ImageStreamTag**.

**name**

Specifies the name of an image stream tag.

**namespace**

Specifies the namespace of the object. This field is optional.

**fieldPath**

Specifies 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**.

**paused**

Specifies whether or not the trigger is paused. This field is optional, and defaults to the value **false**. Set the value 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

To enable automatic updates for your deployed applications managed by Kubernetes, use the command-line interface (CLI) to set an image stream change trigger on Kubernetes resources. This ensures that resources, like **Deployments** and **StatefulSets**, are automatically invoked when a new version of an upstream image is available.

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
```

#### Example deployment with trigger annotation

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  annotations:
    image.openshift.io/triggers: '[{"from":
{"kind":"ImageStreamTag","name":"example:latest"},"fieldPath":"spec.template.spec.containers[
?(@.name==\"container\")].image"}]'
# ...
```

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 (CLASSIC)

You can configure an image registry to store and serve container images.

## 9.1. IMAGE CONTROLLER CONFIGURATION PARAMETERS

You can configure certain parameters that handle images cluster-wide in the **spec** of the **image.config.openshift.io/cluster** resource.

NOTE

The following non-configurable parameters are not listed in the table:

- **DisableScheduledImport**
- **MaxImagesBulkImportedPerRepository**
- **MaxScheduledImportsPerMinute**
- **ScheduledImageImportMinimumIntervalSeconds**
- **InternalRegistryHostname**

Table 9.1. Image controller configuration parameters

| Field name | Description |
|---|---|
| **kind.Image** | Holds cluster-wide information about how to handle images. The canonical, and only valid name for this CR is **cluster**. |
| **allowedRegistriesForImport** | 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.<br><br>Every element of this list contains a location of the registry specified by the registry domain name.<br><br>**domainName**: Specifies a domain name for the registry. If the registry uses a non-standard **80** or **443** port, the port should be included in the domain name as well.<br><br>**insecure**: Insecure indicates whether the registry is secure or insecure. By default, if not otherwise specified, the registry is assumed to be secure. |

| Field name | Description |
| --- | --- |
| **additionalTrustedCA** | 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.<br><br>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. |
| **externalRegistryHostnames** | Provides the hostnames 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. |
| **registrySources** | Contains configuration that determines how the container runtime should treat individual registries when accessing images for builds and pods. For example, whether or not to allow insecure access. It does not contain configuration for the internal cluster registry.<br><br>**insecureRegistries**: Registries that do not have a valid TLS certificate or only support HTTP connections. To specify all subdomains, add the asterisk (*****) wildcard character as a prefix to the domain name. For example, ***.example.com**. You can specify an individual repository within a registry. For example: **reg1.io/myrepo/myapp:latest**.<br><br>**blockedRegistries**: Registries for which image pull and push actions are denied. To specify all subdomains, add the asterisk (*****) wildcard character as a prefix to the domain name. For example, ***.example.com**. You can specify an individual repository within a registry. For example: **reg1.io/myrepo/myapp:latest**. All other registries are allowed.<br><br>**allowedRegistries**: Registries for which image pull and push actions are allowed. To specify all subdomains, add the asterisk (*****) wildcard character as a prefix to the domain name. For example, ***.example.com**. You can specify an individual repository within a registry. For example: **reg1.io/myrepo/myapp:latest**. All other registries are blocked.<br><br>**containerRuntimeSearchRegistries**: Registries for which image pull and push actions are allowed using image short names. All other registries are blocked.<br><br>You can set either **blockedRegistries** or **allowedRegistries**, but not both. |

| Field name | Description |
| --- | --- |
| **imageStreamImportMode** | Controls the import mode behavior of image streams.<br><br>You must enable the **TechPreviewNoUpgrade** feature set in the **FeatureGate** custom resource (CR) to enable the **imageStreamImportMode** feature. For more information about feature gates, see "Understanding feature gates".<br><br>You can set the **imageStreamImportMode** field to either of the following values:<br><br>- **Legacy**: Indicates that the legacy behavior must be used. The legacy behavior discards the manifest list and imports a single sub-manifest. In this case, the platform is chosen in the following order of priority:<br>  1. Tag annotations: Determining the platform by using the platform-specific annotations in the image tags.<br>  2. Control plane architecture or the operating system: Selecting the platform based on the architecture or the operating system of the control plane.<br>  3. **linux/amd64**: If no platform is selected by the preceeding methods, the **linux/amd64** platform is selected.<br>  4. The first manifest in the list is selected.<br>- **PreserveOriginal**: Indicates that the original manifest is preserved. The manifest list and its sub-manifests are imported.<br><br>If you specify a value for this field, the value is applied to the newly created image stream tags that do not already have this value manually set.<br><br>If you do not configure this field, the behavior is decided based on the payload type advertised by the **ClusterVersion** status. In this case, the platform is chosen as follows:<br><br>- The single architecture payload implies that the **Legacy** mode is applicable.<br>- The multi payload implies that the **PreserveOriginal** mode is applicable.<br><br>For information about importing manifest lists, see "Working with manifest lists".<br><br>  <br><br>IMPORTANT<br><br>**imageStreamImportMode** is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.<br><br>For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope. |

| Field name | Description |
|---|---|

**WARNING**

When you define the **allowedRegistries** parameter, all registries, including **registry.redhat.io**, **quay.io**, and the default OpenShift image registry, are blocked unless explicitly listed. You must add all of the registries that your payload images require to the **allowedRegistries** list. For example, list **registry.redhat.io**, **quay.io**, and the **internalRegistryHostname** registries. For disconnected clusters, you must also add your mirror registries. Otherwise, you risk pod failure.

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

**Table 9.2. Image controller status field parameters**

| Parameter | Description |
|---|---|
| **internalRegistryHostname** | Set by the Image Registry Operator, which controls the **internalRegistryHostname**. It sets the hostname for the default OpenShift image registry. The value must be in **hostname[:port]** format. For backward compatibility, you can still use the **OPENSHIFT_DEFAULT_REGISTRY** environment variable, but this setting overrides the environment variable. |
| **externalRegistryHostnames** | Set by the Image Registry Operator, provides the external hostnames 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. MACHINE CONFIG OPERATOR BEHAVIOR AND REGISTRY CHANGES

The Machine Config Operator (MCO) watches the **image.config.openshift.io/cluster** custom resource (CR) for any changes to registries and takes specific steps when the registry changes.

When changes to the registry are applied to the **image.config.openshift.io/cluster** CR, the MCO performs the following sequential actions:

1. Cordons the node; certain parameters result in drained nodes, and others do not
2. Applies changes by restarting CRI-O
3. Uncordons the node

**NOTE**

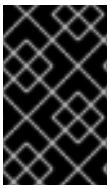
The MCO does not restart nodes when it detects changes. During this period, you might experience service unavailability.

### 9.2.1. When allowing and blocking registry sources

The MCO watches the **image.config.openshift.io/cluster** resource for any changes to the registries. When the MCO detects a change, it triggers a rollout on nodes in machine config pool (MCP). The allowed registries list is used to update the image signature policy in the **/etc/containers/policy.json** file on each node. Changes to the **/etc/containers/policy.json** file do not require the node to drain.

### 9.2.2. When using the containerRuntimeSearchRegistries parameter

After the nodes return to the **Ready** state, if the **containerRuntimeSearchRegistries** parameter is added, the MCO creates a file in the **/etc/containers/registries.conf.d** directory on each node with the listed registries. The file overrides the default list of unqualified search registries in the **/etc/containers/registries.conf** file. There is no way to fall back to the default list of unqualified search registries.



IMPORTANT

The **containerRuntimeSearchRegistries** parameter works only with the Podman and CRI-O container engines. The registries in the list can be used only in pod specs, not in builds and image streams.

## 9.3. CONFIGURING IMAGE REGISTRY SETTINGS

You can configure image registry settings by editing the **image.config.openshift.io/cluster** custom resource (CR).

**Procedure**

1. Edit the **image.config.openshift.io/cluster** CR by running the following command:

```
$ 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
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:
    - domainName: quay.io
      insecure: false
  additionalTrustedCA:
    name: myconfigmap
  registrySources:
    allowedRegistries:
    - example.com
```

```
    - quay.io
    - registry.redhat.io
    - image-registry.openshift-image-registry.svc:5000
    - reg1.io/myrepo/myapp:latest
   insecureRegistries:
   - insecure.com
status:
  internalRegistryHostname: image-registry.openshift-image-registry.svc:5000
```

NOTE

When you use the **allowedRegistries**, **blockedRegistries**, or **insecureRegistries** parameter, you can specify an individual repository within a registry. For example: **reg1.io/myrepo/myapp:latest**.

Avoid insecure external registries to reduce possible security risks.

**Verification**

1. To verify your changes, list your nodes by running the following command:

```
$ oc get nodes
```

**Example output**

```
NAME                                         STATUS                     ROLES                AGE   VERSION
ip-10-0-137-182.us-east-2.compute.internal   Ready,SchedulingDisabled   worker
65m   v1.31.3
ip-10-0-139-120.us-east-2.compute.internal   Ready,SchedulingDisabled   control-plane
74m   v1.31.3
ip-10-0-176-102.us-east-2.compute.internal   Ready                      control-plane        75m
v1.31.3
ip-10-0-188-96.us-east-2.compute.internal    Ready                      worker               65m
v1.31.3
ip-10-0-200-59.us-east-2.compute.internal    Ready                      worker               63m
v1.31.3
ip-10-0-223-123.us-east-2.compute.internal   Ready                      control-plane        73m
v1.31.3
```

### 9.3.1. Adding specific registries to an allowlist

You can add an allowlist of registries, or an individual repository, within a registry 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 your allowlist are blocked.

**WARNING**

When you define the **allowedRegistries** parameter, all registries, including **registry.redhat.io**, **quay.io**, and the default OpenShift image registry, are blocked unless explicitly listed. You must add all of the registries that your payload images require to the **allowedRegistries** list. For example, list **registry.redhat.io**, **quay.io**, and the **internalRegistryHostname** registries. For disconnected clusters, you must also add your mirror registries. Otherwise, you risk pod failure.

**Procedure**

- Edit the **image.config.openshift.io/cluster** custom resource by running the following command:

```
$ 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:
    allowedRegistries:
    - example.com
    - quay.io
    - registry.redhat.io
    - reg1.io/myrepo/myapp:latest
    - image-registry.openshift-image-registry.svc:5000
status:
  internalRegistryHostname: image-registry.openshift-image-registry.svc:5000
```

1. After you make your configuration updates, list your nodes by running the following command:

```
$ oc get nodes
```

Example output

```
NAME              STATUS   ROLES                AGE   VERSION
<node_name>       Ready    control-plane,master   37m   v1.27.8+4fab27b
```

2. Enter debug mode on the node by running the following command:

```
$ oc debug node/<node_name>
```

Replace <node_name> with the name of your node.

3. When prompted, enter **chroot /host** into the terminal:

```
sh-4.4# chroot /host
```

### Verification

1. Check that the registries are in the policy file by running the following command:

```
sh-5.1# cat /etc/containers/policy.json | jq '.'
```

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

#### 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
```

### 9.3.2. Blocking specific registries

You can block any registry, or an individual repository, within a 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. Payload images within your environment require access to these registries.

**Procedure**

- Edit the **image.config.openshift.io/cluster** custom resource by running the following command:

```
$ 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:
    blockedRegistries:
    - untrusted.com
    - reg1.io/myrepo/myapp:latest
status:
  internalRegistryHostname: image-registry.openshift-image-registry.svc:5000
```

You cannot set both the **blockedRegistries** and **allowedRegistries** parameters. You must select one or the other.

1. Get a list of your nodes by running the following command:

```
$ oc get nodes
```

Example output

```
NAME              STATUS   ROLES                AGE   VERSION
<node_name>       Ready    control-plane,master   37m   v1.27.8+4fab27b
```

2. Run the following command to enter debug mode on the node:

```
$ oc debug node/<node_name>
```

Replace <node_name> with the name of the node you want details about.

3. When prompted, enter **chroot /host** into the terminal:

```
sh-4.4# chroot /host
```

### Verification

1. Verify that the registries are in the policy file by running the following command:

```
sh-5.1# cat etc/containers/registries.conf
```

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

#### Example output

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

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

### 9.3.3. Blocking a payload registry

In a mirroring configuration, you can block upstream payload registries in a disconnected environment by using a **ImageContentSourcePolicy** (ICSP) object. The following example procedure demonstrates how to block the **quay.io/openshift-payload** payload registry.

#### Procedure

1. Create the mirror configuration using an **ImageContentSourcePolicy** (ICSP) object to mirror the payload to a registry in your instance. The following example ICSP file mirrors the payload **internal-mirror.io/openshift-payload**:

```
apiVersion: operator.openshift.io/v1alpha1
kind: ImageContentSourcePolicy
metadata:
  name: my-icsp
spec:
  repositoryDigestMirrors:
  - mirrors:
    - internal-mirror.io/openshift-payload
    source: quay.io/openshift-payload
```

2. After the object deploys onto your nodes, verify that the mirror configuration is set by checking the **/etc/containers/registries.conf** custom resource (CR):

   #### Example output

```
[[registry]]
  prefix = ""
  location = "quay.io/openshift-payload"
  mirror-by-digest-only = true

[[registry.mirror]]
  location = "internal-mirror.io/openshift-payload"
```

3. Use the following command to edit the **image.config.openshift.io** CR:

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

4. To block the payload registry, add the following configuration to the **image.config.openshift.io** CR:

```
spec:
  registrySources:
    blockedRegistries:
     - quay.io/openshift-payload
```

#### Verification

- Verify that the upstream payload registry is blocked by checking the **/etc/containers/registries.conf** file on the node.

**Example /etc/containers/registries.conf file**

```
[[registry]]
  prefix = ""
  location = "quay.io/openshift-payload"
  blocked = true
  mirror-by-digest-only = true

[[registry.mirror]]
  location = "internal-mirror.io/openshift-payload"
```

### 9.3.4. Allowing insecure registries

You can add insecure registries, or an individual repository, within a 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. Registries that do not use valid SSL certificates or do not require HTTPS connections are considered insecure.

**IMPORTANT**

Avoid insecure external registries to reduce possible security risks.

+ :leveloffset: +1

**WARNING**

When you define the **allowedRegistries** parameter, all registries, including **registry.redhat.io**, **quay.io**, and the default OpenShift image registry, are blocked unless explicitly listed. You must add all of the registries that your payload images require to the **allowedRegistries** list. For example, list **registry.redhat.io**, **quay.io**, and the **internalRegistryHostname** registries. For disconnected clusters, you must also add your mirror registries. Otherwise, you risk pod failure.

**Procedure**

- Edit the **image.config.openshift.io/cluster** custom resource (CR) by running the following command:

```
$ 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:
    insecureRegistries:
    - insecure.com
    - reg4.io/myrepo/myapp:latest
    allowedRegistries:
    - example.com
    - quay.io
    - registry.redhat.io
    - insecure.com
    - reg4.io/myrepo/myapp:latest
    - image-registry.openshift-image-registry.svc:5000
status:
  internalRegistryHostname: image-registry.openshift-image-registry.svc:5000
```

### Verification

- Check that the registries are added to the policy file by running the following command on a node:

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

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

#### Example output

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

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

## 9.4. ABOUT ADDING REGISTRIES THAT ALLOW IMAGE SHORT NAMES

With an image short name, you can search for images without including the fully qualified domain name in the pull **spec** parameter.

For example, you could use **rhel7/etcd** instead of **registry.access.redhat.com/rhe7/etcd**. You can add registries to search for an image short name by editing the **image.config.openshift.io/cluster** custom resource (CR).

You might use short names in situations where using the full path is not practical. For example, if your cluster references multiple internal registries whose DNS changes often, you would need to update the fully qualified domain names in your pull specs with each change. In this case, using an image short name might be beneficial.

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 **containerRuntimeSearchRegistries** parameter, when pulling an image with a short name, the container runtime searches those registries.

### 9.4.1. When not to use image short names

To avoid deployment failures and security risks when using public registries in OpenShift Container Platform, use fully-qualified image names instead of short names. Short names work with Red Hat internal or private registries, but public registries that require authentication might not deploy images with short names.

You cannot list multiple public registries under the **containerRuntimeSearchRegistries** parameter if each public registry requires different credentials and a cluster does not list the public registry in the global pull secret.

For a public registry that requires authentication, you can use an image short name only if the registry has its credentials stored in the global pull secret.

WARNING

If you list public registries under the **containerRuntimeSearchRegistries** parameter (including the **registry.redhat.io**, **docker.io**, and **quay.io** registries), you expose your credentials to all the registries on the list, and you risk network and registry attacks. Because you can only have one pull secret for pulling images, as defined by the global pull secret, that secret is used to authenticate against every registry in that list. Therefore, if you include public registries in the list, you introduce a security risk.

### 9.4.2. Adding registries that allow image short names

You can add registries to search for an image short name 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.

WARNING

When you define the **allowedRegistries** parameter, all registries, including **registry.redhat.io**, **quay.io**, and the default OpenShift image registry, are blocked unless explicitly listed. You must add all of the registries that your payload images require to the **allowedRegistries** list. For example, list **registry.redhat.io**, **quay.io**, and the **internalRegistryHostname** registries. For disconnected clusters, you must also add your mirror registries. Otherwise, you risk pod failure.

**Procedure**

- 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
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:
    - domainName: quay.io
      insecure: false
  additionalTrustedCA:
    name: myconfigmap
  registrySources:
    containerRuntimeSearchRegistries:
    - reg1.io
    - reg2.io
    - reg3.io
    allowedRegistries:
    - example.com
    - quay.io
    - registry.redhat.io
    - reg1.io
    - reg2.io
    - reg3.io
    - image-registry.openshift-image-registry.svc:5000
...
status:
  internalRegistryHostname: image-registry.openshift-image-registry.svc:5000
```

1. Get a list of your nodes by running the following command:

```
$ oc get nodes
```

Example output

```
NAME              STATUS   ROLES                AGE   VERSION
<node_name>       Ready    control-plane,master   37m   v1.27.8+4fab27b
```

2. Run the following command to enter debug mode on the node:

```
$ oc debug node/<node_name>
```

3. When prompted, enter **chroot /host** into the terminal:

```
sh-4.4# chroot /host
```

**Verification**

1. Verify that registries are added to the policy file by running the following command:

```
sh-5.1# cat /etc/containers/registries.conf.d/01-image-searchRegistries.conf
```

**Example output**

```
unqualified-search-registries = ['reg1.io', 'reg2.io', 'reg3.io']
```

### 9.4.3. Configuring additional trust stores for image registry access

You can add references to a config map that has additional certificate authorities (CAs) to be trusted during image registry access to the **image.config.openshift.io/cluster** custom resource (CR).

**Prerequisites**

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

**Procedure**

1. Create a config map in the **openshift-config** namespace, then and use the config map name in the **AdditionalTrustedCA** parameter of the **image.config.openshift.io** CR. This adds CAs that should be trusted when the cluster contacts external image registries.

**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: |
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
```

where:

**data:registry.example.com:**

An example hostname of a registry for which this CA is to be trusted.

**data:registry-with-port.example.com..5000:**

An example hostname of a registry with the port for which this CA is to be trusted. If the registry has a port, such as **registry-with-port.example.com:5000**, **:** must be replaced with **..**. The PEM certificate content is the value for each additional registry CA to trust.

2. Optional. Configure an additional CA by running the following command:

```
$ 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
```

## 9.5. UNDERSTANDING IMAGE REGISTRY REPOSITORY MIRRORING

By setting up container registry repository mirroring, you can perform the following tasks:

- 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.

Repository mirroring in OpenShift Container Platform includes the following attributes:

- Image pulls are resilient to registry downtimes.
- Clusters in disconnected environments 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.

You can set up repository mirroring 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 data center that is in a disconnected environment.
- After OpenShift Container Platform installation:
  If you did not configure mirroring during OpenShift Container Platform installation, you can do so postinstallation by using any of the following custom resource (CR) objects:
  - **ImageDigestMirrorSet** (IDMS). This object allows you to pull images from a mirrored registry by using digest specifications. The IDMS CR enables you to set a fall back policy that allows or stops continued attempts to pull from the source registry if the image pull fails.

- **ImageTagMirrorSet** (ITMS). This object allows you to pull images from a mirrored registry by using image tags. The ITMS CR enables you to set a fall back policy that allows or stops continued attempts to pull from the source registry if the image pull fails.
- **ImageContentSourcePolicy** (ICSP). This object allows you to pull images from a mirrored registry by using digest specifications. The ICSP CR always falls back to the source registry if the mirrors do not work.

IMPORTANT

Using an **ImageContentSourcePolicy** (ICSP) object to configure repository mirroring is a deprecated feature. Deprecated functionality is still included in OpenShift Container Platform and continues to be supported. It will be removed in a future release and is not recommended for new deployments.

If you have existing YAML files that you used to create **ImageContentSourcePolicy** objects, you can use the **oc adm migrate icsp** command to convert those files to a **ImageDigestMirrorSet** YAML files. For more information, see "Converting ImageContentSourcePolicy (ICSP) files for image registry repository mirroring".

Each of these custom resource objects identify the following information:

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

Note the following actions and how they affect node drain behavior:

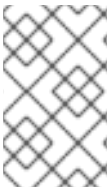
- If you create an IDMS or ICSP CR object, the MCO does not drain or reboot the node.
- If you create an ITMS CR object, the MCO drains and reboots the node.
- If you delete an ITMS, IDMS, or ICSP CR object, the MCO drains and reboots the node.
- If you modify an ITMS, IDMS, or ICSP CR object, the MCO drains and reboots the node.

IMPORTANT

- When the MCO detects any of the following changes, it applies the update without draining or rebooting the node:
  - Changes to the SSH key in the **spec.config.passwd.users.sshAuthorizedKeys** parameter of a machine config.
  - Changes to the global pull secret or pull secret in the **openshift-config** namespace.
  - Automatic rotation of the **/etc/kubernetes/kubelet-ca.crt** certificate authority (CA) by the Kubernetes API Server Operator.
- When the MCO detects changes to the **/etc/containers/registries.conf** file, such as editing an **ImageDigestMirrorSet**, **ImageTagMirrorSet**, or **ImageContentSourcePolicy** object, it drains the corresponding nodes, applies the changes, and uncordons the nodes. The node drain does not happen for the following changes:
  - The addition of a registry with the **pull-from-mirror = "digest-only"** parameter set for each mirror.
  - The addition of a mirror with the **pull-from-mirror = "digest-only"** parameter set in a registry.
  - The addition of items to the **unqualified-search-registries** list.

For new clusters, you can use IDMS, ITMS, and ICSP CRs objects as needed. However, using IDMS and ITMS is recommended.

If you upgraded a cluster, any existing ICSP objects remain stable, and both IDMS and ICSP objects are supported. Workloads that use ICSP objects continue to function as expected. However, if you want to take advantage of the fallback policies introduced in the IDMS CRs, you can migrate current workloads to IDMS objects by using the **oc adm migrate icsp** command as shown in the **Converting ImageContentSourcePolicy (ICSP) files for image registry repository mirroring** section that follows. Migrating to IDMS objects does not require a cluster reboot.



NOTE

If your cluster uses an **ImageDigestMirrorSet**, **ImageTagMirrorSet**, or **ImageContentSourcePolicy** object to configure repository mirroring, you can use only global pull secrets for mirrored registries. You cannot add a pull secret to a project.

### 9.5.1. Configuring image registry repository mirroring

You can create postinstallation mirror configuration custom resources (CR) to redirect image pull requests from a source image registry to a mirrored image registry.

**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. You can copy images from one repository to another and also automatically sync those repositories repeatedly over time by using Red Hat Quay.

     - Red Hat Quay Repository Mirroring

   - Using a tool such as **skopeo** to copy images manually from the source repository to the mirrored repository.
     For example, after installing the skopeo RPM package on a {op-system-base-full system}, use the **skopeo** command as shown in the following example:

     ```
     $ skopeo copy --all \
     docker://registry.access.redhat.com/ubi9/ubi-minimal:latest@sha256:5cf... \
     docker://example.io/example/ubi-minimal
     ```

     In this example, you have a container image registry named **example.io** and image repository named **example**. You want to copy the **ubi9/ubi-minimal** image from **registry.access.redhat.com** to **example.io**. After you create the mirrored registry, you can configure your OpenShift Container Platform cluster to redirect requests made to the source repository to the mirrored repository.

2. Create a postinstallation mirror configuration custom resource (CR), by using one of the following examples:

   - Create an **ImageDigestMirrorSet** or **ImageTagMirrorSet** CR, as needed, replacing the source and mirrors with your own registry and repository pairs and images:

     ```
     apiVersion: config.openshift.io/v1
     kind: ImageDigestMirrorSet
     metadata:
       name: ubi9repo
     spec:
       imageDigestMirrors:
       - mirrors:
         - example.io/example/ubi-minimal
         - example.com/example2/ubi-minimal
         source: registry.access.redhat.com/ubi9/ubi-minimal
         mirrorSourcePolicy: AllowContactingSource
       - mirrors:
         - mirror.example.com/redhat
         source: registry.example.com/redhat
         mirrorSourcePolicy: AllowContactingSource
       - mirrors:
         - mirror.example.com
         source: registry.example.com
         mirrorSourcePolicy: AllowContactingSource
       - mirrors:
         - mirror.example.net/image
         source: registry.example.com/example/myimage
         mirrorSourcePolicy: AllowContactingSource
       - mirrors:
         - mirror.example.net
         source: registry.example.com/example
     ```

```
    mirrorSourcePolicy: AllowContactingSource
  - mirrors:
    - mirror.example.net/registry-example-com
    source: registry.example.com
    mirrorSourcePolicy: AllowContactingSource
```

- Create an **ImageContentSourcePolicy** custom resource, replacing the source and mirrors with your own registry and repository pairs and images:

```
apiVersion: operator.openshift.io/v1alpha1
kind: ImageContentSourcePolicy
metadata:
  name: mirror-ocp
spec:
  repositoryDigestMirrors:
  - mirrors:
    - mirror.registry.com:443/ocp/release
    source: quay.io/openshift-release-dev/ocp-release
  - mirrors:
    - mirror.registry.com:443/ocp/release
    source: quay.io/openshift-release-dev/ocp-v4.0-art-dev
```

where:

**- mirror.registry.com:443/ocp/release**

Specifies the name of the mirror image registry and repository.

**source: quay.io/openshift-release-dev/ocp-release**

Specifies the online registry and repository containing the content that is mirrored.

3. Create the new object by running the following command:

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

After the object is created, the Machine Config Operator (MCO) drains the nodes for **ImageTagMirrorSet** objects only. The MCO does not drain the nodes for **ImageDigestMirrorSet** and **ImageContentSourcePolicy** objects.

4. 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   v1.31.3
ip-10-0-138-148.ec2.internal   Ready                         master   11m  v1.31.3
ip-10-0-139-122.ec2.internal   Ready                         master   11m  v1.31.3
ip-10-0-147-35.ec2.internal    Ready                        worker   7m   v1.31.3
ip-10-0-153-12.ec2.internal    Ready                        worker   7m   v1.31.3
ip-10-0-154-10.ec2.internal    Ready                        master   11m  v1.31.3
```

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. Change your root directory to **/host**:

```
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
```

The following output represents a **registries.conf** file where postinstallation mirror configuration CRs are applied.

**Example output**

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

[[registry]]
  prefix = ""
  location = "registry.access.redhat.com/ubi9/ubi-minimal"

  [[registry.mirror]]
    location = "example.io/example/ubi-minimal"
    pull-from-mirror = "digest-only"

  [[registry.mirror]]
    location = "example.com/example/ubi-minimal"
    pull-from-mirror = "digest-only"

[[registry]]
  prefix = ""
  location = "registry.example.com"

  [[registry.mirror]]
    location = "mirror.example.net/registry-example-com"
    pull-from-mirror = "digest-only"

[[registry]]
  prefix = ""
  location = "registry.example.com/example"

  [[registry.mirror]]
    location = "mirror.example.net"
    pull-from-mirror = "digest-only"

[[registry]]
```

```
    prefix = ""
    location = "registry.example.com/example/myimage"

    [[registry.mirror]]
      location = "mirror.example.net/image"
      pull-from-mirror = "digest-only"

  [[registry]]
    prefix = ""
    location = "registry.example.com"

    [[registry.mirror]]
      location = "mirror.example.com"
      pull-from-mirror = "digest-only"

  [[registry]]
    prefix = ""
    location = "registry.example.com/redhat"

    [[registry.mirror]]
      location = "mirror.example.com/redhat"
      pull-from-mirror = "digest-only"
  [[registry]]
    prefix = ""
    location = "registry.access.redhat.com/ubi9/ubi-minimal"
    blocked = true

    [[registry.mirror]]
      location = "example.io/example/ubi-minimal-tag"
      pull-from-mirror = "tag-only"
```

where: **[[registry]].location = "registry.access.redhat.com/ubi9/ubi-minimal"**:: The repository listed in a pull spec. **[[registry.mirror]].location = "example.io/example/ubi-minimal"**:: Indicates the mirror for that repository. **[[registry.mirror]].pull-from-mirror = "digest-only"**:: Means that the image pull from the mirror is a digest reference image. **[[registry]].blocked = true**:: Indicates that the **NeverContactSource** parameter is set for this repository. **[[registry.mirror]].pull-from-mirror = "tag-only"**:: Indicates that the image pull from the mirror is a tag reference image.

e. Pull an image to the node from the source and check if it is resolved by the mirror.

```
sh-4.2# podman pull --log-level=debug registry.access.redhat.com/ubi9/ubi-minimal@sha256:5cf...
```

### Troubleshooting

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.

### 9.5.2. Image registry repository mirroring configuration parameters

You can use the following table for information about parameters when configuring your image repository for mirroring.

| Parameter | Values and Information |
|---|---|
| **apiVersion:** | Required. The value must be **config.openshift.io/v1** API. |
| **kind:** | The kind of object according to the pull type. The **ImageDigestMirrorSet** type pulls a digest reference image The **ImageTagMirrorSet** type pulls a tag reference image. |
| **spec: imageDigestMirrors:** | The type of image pull method. Use **imageDigestMirrors`for an `ImageDigestMirrorSet** CR. Use **imageTagMirrors** for an **ImageTagMirrorSet** CR. |
| **- mirrors: - example.io/example/ubi-minimal** | The name of the mirrored image registry and repository. |
| **- mirrors: - example.com/example2/ubi-minimal** | The value of this parameter is the name of a secondary mirror repository for each target repository. If one mirror is down the target repository can use the secondary mirror. |
| **source: registry.access.redhat.com/ubi9/ubi-minimal** | The registry and repository source. The source is the repository that is listed in an image pull specification. |
| **mirrorSourcePolicy: AllowContactingSource** | Optional parameter that indicates the fallback policy if the image pull fails. The **AllowContactingSource** value allows continued attempts to pull the image from the source repository. Default value. **NeverContactSource** prevents continued attempts to pull the image from the source repository. |
| **source: registry.example.com/redhat**: An optional parameter that indicates a namespace inside a registry. Setting a namespace inside a registry allows use of any image in that namespace. If you use a registry domain as a source, the object applies to all of the repositories from the registry. | **source: registry.example.com** |

| Parameter | Values and Information |
| --- | --- |
| Optional parameter that indicates a registry. Allows us of any image in that registry. If you specify a registry name, the object applies to all repositories from a source registry to a mirror registry. | **source: registry.example.com/example/myimage** |
| Pulls the image **registry.example.com/example/myimage@sha256:...** from the mirror **mirror.example.net/image@sha256:..**. | **source: registry.example.com/example** |
| Pulls the image **registry.example.com/example/image@sha256:...** in the source registry namespace from the mirror **mirror.example.net/image@sha256:...**. | **source: registry.example.com** |

### 9.5.3. Converting ImageContentSourcePolicy (ICSP) files for image registry repository mirroring

Using an **ImageContentSourcePolicy** (ICSP) object to configure repository mirroring is a deprecated feature.

This functionality is still included in OpenShift Container Platform and continues to be supported; however, it will be removed in a future release of this product and is not recommended for new deployments.

ICSP objects are being replaced by **ImageDigestMirrorSet** and **ImageTagMirrorSet** objects to configure repository mirroring. If you have existing YAML files that you used to create **ImageContentSourcePolicy** objects, you can use the **oc adm migrate icsp** command to convert those files to an **ImageDigestMirrorSet** YAML file. The command updates the API to the current version, changes the **kind** value to **ImageDigestMirrorSet**, and changes **spec.repositoryDigestMirrors** to **spec.imageDigestMirrors**. The rest of the file is not changed.

Because the migration does not change the **registries.conf** file, the cluster does not need to reboot.

For more information about **ImageDigestMirrorSet** or **ImageTagMirrorSet** objects, see "Configuring image registry repository mirroring" in the previous section.

**Prerequisites**

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

- Ensure that you have **ImageContentSourcePolicy** objects on your cluster.

**Procedure**

1. Use the following command to convert one or more **ImageContentSourcePolicy** YAML files to an **ImageDigestMirrorSet** YAML file:

```
$ oc adm migrate icsp <file_name>.yaml <file_name>.yaml <file_name>.yaml --dest-dir
<path_to_the_directory>
```

where:

**<file_name>**

Specifies the name of the source **ImageContentSourcePolicy** YAML. You can list multiple file names.

**--dest-dir**

Optional: Specifies a directory for the output **ImageDigestMirrorSet** YAML. If unset, the file is written to the current directory.
For example, the following command converts the **icsp.yaml** and **icsp-2.yaml** file and saves the new YAML files to the **idms-files** directory.

```
$ oc adm migrate icsp icsp.yaml icsp-2.yaml --dest-dir idms-files
```

**Example output**

```
wrote ImageDigestMirrorSet to idms-
files/imagedigestmirrorset_ubi8repo.5911620242173376087.yaml
wrote ImageDigestMirrorSet to idms-
files/imagedigestmirrorset_ubi9repo.6456931852378115011.yaml
```

2. Create the CR object by running the following command:

```
$ oc create -f <path_to_the_directory>/<file-name>.yaml
```

where:

**<path_to_the_directory>**

Specifies the path to the directory, if you used the **--dest-dir** flag.

**<file_name>**

Specifies the name of the **ImageDigestMirrorSet** YAML.

3. Remove the ICSP objects after the IDMS objects are rolled out.

## 9.6. ADDITIONAL RESOURCES

- Working with manifest lists
- Understanding feature gates
- Updating the global cluster pull secret

# CHAPTER 10. USING IMAGES

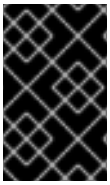
## 10.1. USING IMAGES OVERVIEW

To build and deploy containerized applications in OpenShift Container Platform, you can use Source-to-Image (S2I), database, and other container images. These images provide the base components you need to run applications on your cluster.

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.

## 10.2. SOURCE-TO-IMAGE

To create containerized applications in OpenShift Container Platform without manually configuring runtime environments, you can use Source-to-Image (S2I) images. S2I images are runtime base images for languages like Node.js, Python, and Java that you can insert your code into.

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.

You can use the Introduction to source-to-image for OpenShift documentation as a reference for runtime environments that use Java.

S2I images are also available though the Cluster Samples Operator.

### 10.2.1. Accessing S2I builder images in the OpenShift Container Platform Developer Console

You can access S2I builder images through the Developer Console in the web console. You need these images to build containerized applications from your source code.

**Procedure**

1. Log in to the OpenShift Container Platform web console using your login credentials. The default view for the OpenShift Container Platform web console is the **Administrator** perspective.

2. Use the perspective switcher to switch to the **Developer** perspective.

3. In the **+Add** view, use the **Project** drop-down list to select an existing project or create a new project.
4. Click **All services** in the **Developer Catalog** tile.
5. Click **Builder Images** under **Type** to see the available S2I images.

### 10.2.2. Source-to-image build process overview

Source-to-image (S2I) is a build process in OpenShift Container Platform that injects your source code into a container image. S2I automates the creation of ready-to-run container images from your application source code without manual configuration.

S2I 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.

### 10.2.3. Additional resources

- Configuring the Cluster Samples Operator
- Using build strategies
- Troubleshooting the Source-to-Image process
- Creating images from source code with source-to-image
- About testing source-to-image images
- Creating images from source code with source-to-image

## 10.3. CUSTOMIZING SOURCE-TO-IMAGE IMAGES

To modify the default assemble and run script behavior in OpenShift Container Platform, you can customize source-to-image (S2I) builder images. You can adapt S2I builders to meet your specific application requirements when the default scripts are not suitable.

### 10.3.1. Invoking scripts embedded in an image

To extend builder image behavior while preserving supported script logic and upgrade compatibility in OpenShift Container Platform, you can start embedded S2I image scripts by creating wrapper scripts. These wrapper scripts run custom logic and then call the default scripts from the image.

**Procedure**

1. Inspect 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
```

2. Create a script that includes an invocation of one of the standard scripts wrapped in other commands:

**.s2i/bin/assemble script**

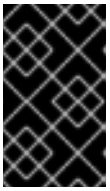
```
#!/bin/bash
echo "Before assembling"

/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
```