Red Hat OpenShift Service on AWS 4

Images

Red Hat OpenShift Service on AWS Images.

Last Updated: 2024-06-06

Red Hat OpenShift Service on AWS 4 Images

Red Hat OpenShift Service on AWS Images.

Legal Notice

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

Abstract

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

Table of Contents

CHAPTER 1. OVERVIEW OF IMAGES 6
1.1. UNDERSTANDING CONTAINERS, IMAGES, AND IMAGE STREAMS 6
1.2. IMAGES 6
1.3. IMAGE REGISTRY 6
1.4. IMAGE REPOSITORY 6
1.5. IMAGE TAGS 6
1.6. IMAGE IDS 7
1.7. CONTAINERS 7
1.8. WHY USE IMAGESTREAMS 7
1.9. IMAGE STREAM TAGS 8
1.10. IMAGE STREAM IMAGES 9
1.11. IMAGE STREAM TRIGGERS 9
1.12. HOW YOU CAN USE THE CLUSTER SAMPLES OPERATOR 9
1.13. ABOUT TEMPLATES 9
1.14. HOW YOU CAN USE RUBY ON RAILS 9

CHAPTER 2. OVERVIEW OF THE CLUSTER SAMPLES OPERATOR 10
2.1. UNDERSTANDING THE CLUSTER SAMPLES OPERATOR 10
2.1.1. Cluster Samples Operator's use of management state 11
2.1.2. Cluster Samples Operator's tracking and error recovery of image stream imports 12
Additional resources 12
2.2. REMOVING DEPRECATED IMAGE STREAM TAGS FROM THE CLUSTER SAMPLES OPERATOR 12
Additional resources 13

CHAPTER 3. USING THE CLUSTER SAMPLES OPERATOR WITH AN ALTERNATE REGISTRY 14
3.1. ABOUT THE MIRROR REGISTRY 14
3.1.1. Preparing the mirror host 14
3.1.2. Installing the OpenShift CLI by downloading the binary 15
Installing the OpenShift CLI on Linux 15
Installing the OpenShift CLI on Windows 15
Installing the OpenShift CLI on macOS 16
3.2. CONFIGURING CREDENTIALS THAT ALLOW IMAGES TO BE MIRRORED 16
3.3. MIRRORING THE RED HAT OPENSHIFT SERVICE ON AWS IMAGE REPOSITORY 18
3.4. USING CLUSTER SAMPLES OPERATOR IMAGE STREAMS WITH ALTERNATE OR MIRRORED REGISTRIES 20
3.4.1. Cluster Samples Operator assistance for mirroring 22

CHAPTER 4. CREATING IMAGES 23
4.1. LEARNING CONTAINER BEST PRACTICES 23
4.1.1. General container image guidelines 23
Reuse images 23
Maintain compatibility within tags 23
Avoid multiple processes 23
Use exec in wrapper scripts 23
Clean temporary files 24
Place instructions in the proper order 24
Mark important ports 25
Set environment variables 25
Avoid default passwords 25
Avoid sshd 25
Use volumes for persistent data 25
4.1.2. Red Hat OpenShift Service on AWS-specific guidelines 26

4.1.2.1. Enable images for source-to-image (S2I) 26
4.1.2.2. Support arbitrary user ids 26
4.1.2.3. Use services for inter-image communication 27
4.1.2.4. Provide common libraries 27
4.1.2.5. Use environment variables for configuration 27
4.1.2.6. Set image metadata 28
4.1.2.7. Clustering 28
4.1.2.8. Logging 28
4.1.2.9. Liveness and readiness probes 28
4.1.2.10. Templates 28
4.2. INCLUDING METADATA IN IMAGES 29
4.2.1. Defining image metadata 29
4.3. CREATING IMAGES FROM SOURCE CODE WITH SOURCE-TO-IMAGE 30
4.3.1. Understanding the source-to-image build process 30
4.3.2. How to write source-to-image scripts 30
4.4. ABOUT TESTING SOURCE-TO-IMAGE IMAGES 33
4.4.1. Understanding testing requirements 33
4.4.2. Generating scripts and tools 33
4.4.3. Testing locally 34
4.4.4. Basic testing workflow 34
4.4.5. Using Red Hat OpenShift Service on AWS for building the image 35

CHAPTER 5. MANAGING IMAGES **36**
5.1. MANAGING IMAGES OVERVIEW 36
5.1.1. Images overview 36
5.2. TAGGING IMAGES 36
5.2.1. Image tags 36
5.2.2. Image tag conventions 36
5.2.3. Adding tags to image streams 37
5.2.4. Removing tags from image streams 38
5.2.5. Referencing images in imagestreams 38
5.3. IMAGE PULL POLICY 39
5.3.1. Image pull policy overview 39
5.4. USING IMAGE PULL SECRETS 40
5.4.1. Allowing pods to reference images across projects 40
5.4.2. Allowing pods to reference images from other secured registries 41
5.4.2.1. Pulling from private registries with delegated authentication 42

CHAPTER 6. MANAGING IMAGE STREAMS **44**
6.1. WHY USE IMAGESTREAMS 44
6.2. CONFIGURING IMAGE STREAMS 45
6.3. IMAGE STREAM IMAGES 46
6.4. IMAGE STREAM TAGS 46
6.5. IMAGE STREAM CHANGE TRIGGERS 47
6.6. WORKING WITH IMAGE STREAMS 47
6.6.1. Getting information about image streams 48
6.6.2. Adding tags to an image stream 49
6.6.3. Adding tags for an external image 50
6.6.4. Updating image stream tags 51
6.6.5. Removing image stream tags 51
6.6.6. Configuring periodic importing of image stream tags 52
6.7. IMPORTING AND WORKING WITH IMAGES AND IMAGE STREAMS 52
6.7.1. Importing images and image streams from private registries 52

6.7.1.1. Allowing pods to reference images from other secured registries 53
6.7.2. Working with manifest lists 54
Limitations 55
6.7.2.1. Configuring periodic importing of manifest lists 55
6.7.2.2. Configuring SSL/TSL when importing manifest lists 55
6.7.3. Specifying architecture for --import-mode 56
6.7.4. Configuration fields for --import-mode 56

CHAPTER 7. USING IMAGE STREAMS WITH KUBERNETES RESOURCES 57
7.1. ENABLING IMAGE STREAMS WITH KUBERNETES RESOURCES 57

CHAPTER 8. TRIGGERING UPDATES ON IMAGE STREAM CHANGES 59
8.1. RED HAT OPENSHIFT SERVICE ON AWS RESOURCES 59
8.2. TRIGGERING KUBERNETES RESOURCES 59
8.3. SETTING THE IMAGE TRIGGER ON KUBERNETES RESOURCES 60

CHAPTER 9. IMAGE CONFIGURATION RESOURCES 61
9.1. IMAGE CONTROLLER CONFIGURATION PARAMETERS 61
9.2. CONFIGURING IMAGE REGISTRY SETTINGS 63
9.2.1. Adding specific registries 65
9.2.2. Blocking specific registries 67
9.2.3. Allowing insecure registries 68
9.2.4. Adding registries that allow image short names 69
9.2.5. Configuring additional trust stores for image registry access 72
9.3. UNDERSTANDING IMAGE REGISTRY REPOSITORY MIRRORING 73
9.3.1. Configuring image registry repository mirroring 74
9.3.2. Converting ImageContentSourcePolicy (ICSP) files for image registry repository mirroring 79

CHAPTER 10. USING TEMPLATES 82
10.1. UNDERSTANDING TEMPLATES 82
10.2. UPLOADING A TEMPLATE 82
10.3. CREATING AN APPLICATION BY USING THE WEB CONSOLE 82
10.4. CREATING OBJECTS FROM TEMPLATES BY USING THE CLI 83
10.4.1. Adding labels 83
10.4.2. Listing parameters 83
10.4.3. Generating a list of objects 84
10.5. MODIFYING UPLOADED TEMPLATES 86
10.6. WRITING TEMPLATES 86
10.6.1. Writing the template description 86
10.6.2. Writing template labels 91
10.6.3. Writing template parameters 91
10.6.4. Writing the template object list 94
10.6.5. Marking a template as bindable 95
10.6.6. Exposing template object fields 95
10.6.7. Waiting for template readiness 97
10.6.8. Creating a template from existing objects 99

CHAPTER 11. USING RUBY ON RAILS 100
11.1. PREREQUISITES 100
11.2. SETTING UP THE DATABASE 100
11.3. WRITING YOUR APPLICATION 101
11.3.1. Creating a welcome page 102
11.3.2. Configuring application for Red Hat OpenShift Service on AWS 102
11.3.3. Storing your application in Git 103

11.4. DEPLOYING YOUR APPLICATION TO RED HAT OPENSHIFT SERVICE ON AWS 104
11.4.1. Creating the database service 104
11.4.2. Creating the frontend service 105
11.4.3. Creating a route for your application 106

CHAPTER 1. OVERVIEW OF IMAGES

1.1. UNDERSTANDING CONTAINERS, IMAGES, AND IMAGE STREAMS

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

1.2. IMAGES

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

You can think of it as a packaging technology. Containers only have access to resources defined in the image unless you give the container additional access when creating it. By deploying the same image in multiple containers across multiple hosts and load balancing between them, Red Hat OpenShift Service on AWS 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 Red Hat OpenShift Service on AWS 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**.

You can create and manage container images.

1.3. IMAGE REGISTRY

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

```
registry.redhat.io
```

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

1.4. IMAGE REPOSITORY

An image repository is a collection of related container images and tags identifying them. For example, the Red Hat OpenShift Service on AWS Jenkins images are in the repository:

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

1.5. IMAGE TAGS

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

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

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

Red Hat OpenShift Service on AWS provides the **oc tag** command, which is similar to the **docker tag** command, but operates on image streams instead of directly on images.

1.6. IMAGE IDS

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

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

1.7. CONTAINERS

The basic units of Red Hat OpenShift Service on AWS applications are called containers. Linux container technologies are lightweight mechanisms for isolating running processes so that they are limited to interacting with only their designated resources. The word container is defined as a specific running or paused instance of a container image.

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

The Linux kernel has been incorporating capabilities for container technologies for years. The Docker project developed a convenient management interface for Linux containers on a host. More recently, the Open Container Initiative has developed open standards for container formats and container runtimes. Red Hat OpenShift Service on AWS 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 Red Hat OpenShift Service on AWS, understanding their capabilities and terminology is important for understanding their role in Red Hat OpenShift Service on AWS and how your applications function inside of containers.

Tools such as podman can be used to replace **docker** command-line tools for running and managing containers directly. Using **podman**, you can experiment with containers separately from Red Hat OpenShift Service on AWS.

1.8. WHY USE IMAGESTREAMS

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

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

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

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

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

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

- Red Hat OpenShift Service on AWS's integrated registry.
- An external registry, for example registry.redhat.io or quay.io.
- Other image streams in the Red Hat OpenShift Service on AWS 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, Red Hat OpenShift Service on AWS 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, Red Hat OpenShift Service on AWS 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.

You can manage image streams, use image streams with Kubernetes resources, and trigger updates on image stream updates.

1.9. IMAGE STREAM TAGS

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

1.10. IMAGE STREAM IMAGES

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

1.11. IMAGE STREAM TRIGGERS

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

1.12. HOW YOU CAN USE THE CLUSTER SAMPLES OPERATOR

During the initial startup, the Operator creates the default samples resource to initiate the creation of the image streams and templates. You can use the Cluster Samples Operator to manage the sample image streams and templates stored in the **openshift** namespace.

As a cluster administrator, you can use the Cluster Samples Operator to:

- Use the Operator with an alternate registry.

1.13. ABOUT TEMPLATES

A template is a definition of an object to be replicated. You can use templates to build and deploy configurations.

1.14. HOW YOU CAN USE RUBY ON RAILS

As a developer, you can use Ruby on Rails to:

- Write your application:
  - Set up a database.
  - Create a welcome page.
  - Configure your application for Red Hat OpenShift Service on AWS.
  - Store your application in Git.
- Deploy your application in Red Hat OpenShift Service on AWS:
  - Create the database service.
  - Create the frontend service.
  - Create a route for your application.

CHAPTER 2. OVERVIEW OF THE CLUSTER SAMPLES OPERATOR

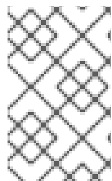
The Cluster Samples Operator, which operates in the **openshift** namespace, installs and updates the Red Hat OpenShift Service on AWS image streams and Red Hat OpenShift Service on AWS templates.

CLUSTER SAMPLES OPERATOR IS BEING DOWNSIZED

- Starting from Red Hat OpenShift Service on AWS 4.13, Cluster Samples Operator is downsized. Cluster Samples Operator will stop providing the following updates for non-Source-to-Image (Non-S2I) image streams and templates:
  - new image streams and templates
  - updates to the existing image streams and templates unless it is a CVE update
- Cluster Samples Operator will provide support for Non-S2I image streams and templates as per the Red Hat OpenShift Service on AWS lifecycle policy dates and support guidelines.
- Cluster Samples Operator will continue to support the S2I builder image streams and templates and accept the updates. S2I image streams and templates include:
  - Ruby
  - Python
  - Node.js
  - Perl
  - PHP
  - HTTPD
  - Nginx
  - EAP
  - Java
  - Webserver
  - .NET
  - Go
- Starting from Red Hat OpenShift Service on AWS 4.16, Cluster Samples Operator will stop managing non-S2I image streams and templates. You can contact the image stream or template owner for any requirements and future plans. In addition, refer to the 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.

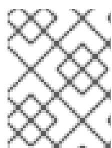


NOTE

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, and the deployment is contained within the **openshift-cluster-samples-operator** namespace.

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



NOTE

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

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

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

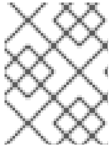
Upon deletion of the samples resource, the Cluster Samples Operator recreates the resource using the default configuration.

2.1.1. Cluster Samples Operator’s use of management state

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

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

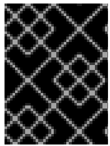
- If the Cluster Samples Operator cannot reach registry.redhat.io after three minutes on initial startup after a clean installation.
- If the Cluster Samples Operator detects it is on an IPv6 network.



NOTE

For Red Hat OpenShift Service on AWS, the default image registry is **registry.access.redhat.com** or **quay.io**.

However, if the Cluster Samples Operator detects that it is on an IPv6 network and an Red Hat OpenShift Service on AWS global proxy is configured, then 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 registry.redhat.io. The Cluster Samples Operator pulls most of the sample image streams and images from registry.redhat.io.

2.1.2. Cluster Samples Operator’s tracking and error recovery of image stream imports

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

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

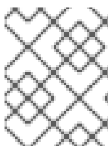
Additional resources

- If the Cluster Samples Operator is removed during installation, you can use the Cluster Samples Operator with an alternate registry so content can be imported, and then set the Cluster Samples Operator to **Managed** to get the samples.

2.2. 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 **oc** CLI.

Procedure

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

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

Example output

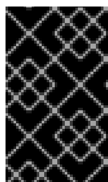
```
Deleted tag default/<image_stream_name:tag>.
```

Additional resources

- For more information about configuring credentials, see Using image pull secrets.

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.



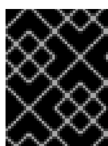
IMPORTANT

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

3.1. ABOUT THE MIRROR REGISTRY

You can mirror the images that are required for Red Hat OpenShift Service on AWS 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 Red Hat OpenShift Service on AWS 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.



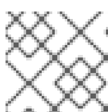
IMPORTANT

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 Red Hat OpenShift Service on AWS clusters.

When you populate your mirror registry with Red Hat OpenShift Service on AWS 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

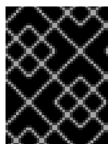
Red Hat does not test third party registries with Red Hat OpenShift Service on AWS.

3.1.1. Preparing the mirror host

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

3.1.2. Installing the OpenShift CLI by downloading the binary

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



IMPORTANT

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

Installing the OpenShift CLI on Linux

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

Procedure

1. Navigate to the Red Hat OpenShift Service on AWS 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 Linux Client** 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>
```

Installing the OpenShift CLI on Windows

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

Procedure

1. Navigate to the Red Hat OpenShift Service on AWS downloads page on the Red Hat Customer Portal.
2. Select the appropriate version from the **Version** drop-down list.
3. Click **Download Now** next to the **OpenShift v4 Windows Client** entry and save the file.

4. Unzip the archive with a ZIP program.

5. Move the **oc** binary to a directory that is on your **PATH**.
   To check your **PATH**, open the command prompt and execute the following command:

   ```
   C:\> path
   ```

Verification

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

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

Installing the OpenShift CLI on macOS

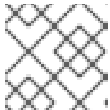
You can install the OpenShift CLI (**oc**) binary on macOS by using the following procedure.

Procedure

1. Navigate to the Red Hat OpenShift Service on AWS downloads page on the Red Hat Customer Portal.

2. Select the appropriate version from the **Version** drop-down list.

3. Click **Download Now** next to the **OpenShift v4 macOS Client** entry and save the file.

   NOTE

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

4. Unpack and unzip the archive.

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

   ```
   $ echo $PATH
   ```

Verification

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

  ```
  $ oc <command>
  ```

3.2. CONFIGURING CREDENTIALS THAT ALLOW IMAGES TO BE MIRRORED

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

Prerequisites

- You configured a mirror registry to use.

Procedure

Complete the following steps on the installation host:

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:

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

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

The contents of the file resemble the following example:

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

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

```
$ echo -n '<user_name>:<password>' | base64 -w0 1
BGVtbYk3ZHAtqXs=
```

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

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

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

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

The file resembles the following example:

```
{
  "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 RED HAT OPENSHIFT SERVICE ON AWS IMAGE REPOSITORY

Mirror the Red Hat OpenShift Service on AWS image repository to your registry to use during cluster installation or upgrade.

Prerequisites

- Your mirror host has access to the internet.
- You configured a mirror registry to use.
- 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

Complete the following steps on the mirror host:

1. Review the Red Hat OpenShift Service on AWS downloads page to determine the version of Red Hat OpenShift Service on AWS that you want to install and determine the corresponding tag on the Repository Tags page.

2. Set the required environment variables:

   a. Export the release version:

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

   For **<release_version>**, specify the tag that corresponds to the version of Red Hat OpenShift Service on AWS to install, such as **4.5.4**.

   b. Export the local registry name and host port:

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

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

   c. Export the local repository name:

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

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

   d. Export the name of the repository to mirror:

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

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

   e. Export the path to your registry pull secret:

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

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

   f. Export the release mirror:

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

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

   g. Export the type of architecture for your cluster:

   ```
   $ ARCHITECTURE=<cluster_architecture> 1
   ```

   1 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> 1
```

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

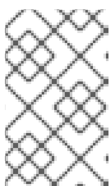
3. Mirror the version images to the mirror registry:

   a. 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.

   b. 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.

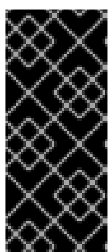
   

   NOTE

   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 by running 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 Red Hat OpenShift Service on AWS that you selected, you must extract the installation program from the mirrored content.

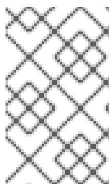
   You must perform this step on a machine with an active internet connection.

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

   ```
   $ openshift-install
   ```

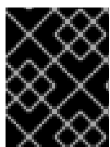
3.4. USING CLUSTER SAMPLES OPERATOR IMAGE STREAMS WITH ALTERNATE OR MIRRORED REGISTRIES

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 have a mirrored registry.

Prerequisites

- Access to the cluster as a user with the **dedicated-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 cluster's image configuration object:

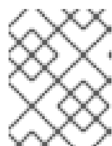
```
$ oc create configmap registry-config --from-
file=${MIRROR_ADDR_HOSTNAME}..5000=$path/ca.crt -n openshift-config
```

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

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

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

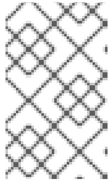
```
$ oc edit configs.samples.operator.openshift.io -n openshift-cluster-samples-operator
```



NOTE

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

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



NOTE

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.

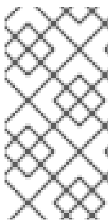
Many of the templates in the **openshift** namespace reference the image streams. So using **Removed** to purge both the image streams and templates will eliminate the possibility of attempts to use them if they are not functional because of any missing image streams.

3.4.1. Cluster Samples Operator assistance for mirroring

During installation, Red Hat OpenShift Service on AWS 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 Red Hat OpenShift Service on AWS, 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 may 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's objects to reach the services they require.

You can use this config map as a reference for which images need to be mirrored 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.

See Using Cluster Samples Operator image streams with alternate or mirrored registries for a detailed procedure.

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 Red Hat OpenShift Service on AWS.

4.1. LEARNING CONTAINER BEST PRACTICES

When creating container images to run on Red Hat OpenShift Service on AWS 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 Red Hat OpenShift Service on AWS.

4.1.1. General container image guidelines

The following guidelines apply when creating a container image in general, and are independent of whether the images are used on Red Hat OpenShift Service on AWS.

Reuse images

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

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

Maintain compatibility within tags

When tagging your own images, try to maintain backwards compatibility within a tag. For example, if you provide an image named **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.

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. Red Hat OpenShift Service on AWS allows you to easily colocate and co-manage related images by grouping them into a single pod.

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

Use exec in wrapper scripts

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

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

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

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

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

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

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

Note that if you instead write:

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

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

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

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

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

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

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

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

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

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

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

Mark important ports

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

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

Set environment variables

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

Avoid default passwords

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

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

Avoid sshd

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

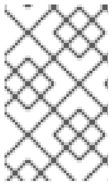
Use volumes for persistent data

Images use a volume for persistent data. This way Red Hat OpenShift Service on AWS 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 Red Hat OpenShift Service on AWS.



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. Red Hat OpenShift Service on AWS-specific guidelines

The following are guidelines that apply when creating container images specifically for use on Red Hat OpenShift Service on AWS.

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, Red Hat OpenShift Service on AWS 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.

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 Red Hat OpenShift Service on AWS 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.

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 Red Hat OpenShift Service on AWS 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 Red Hat OpenShift Service on AWS better consume your container images, allowing Red Hat OpenShift Service on AWS 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 Red Hat OpenShift Service on AWS. 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. Red Hat OpenShift Service on AWS 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

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

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

4.2.1. Defining image metadata

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

Docker documentation for more information on the **LABEL** instruction.

The label names are typically namespaced. The namespace is set accordingly to reflect the project that is going to pick up the labels and use them. For Red Hat OpenShift Service on AWS 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. `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. `LABEL io.openshift.wants   mongodb,redis`

Variable	Description
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. `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**. `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.
	`LABEL io.openshift.min-memory 16Gi`
	`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

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

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

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

4.3.2. How to write source-to-image scripts

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

1. A script specified in the build configuration.

2. A script found in the application source **.s2i/bin** directory.

3. A script found at the default image URL with the **io.openshift.s2i.scripts-url** label.

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

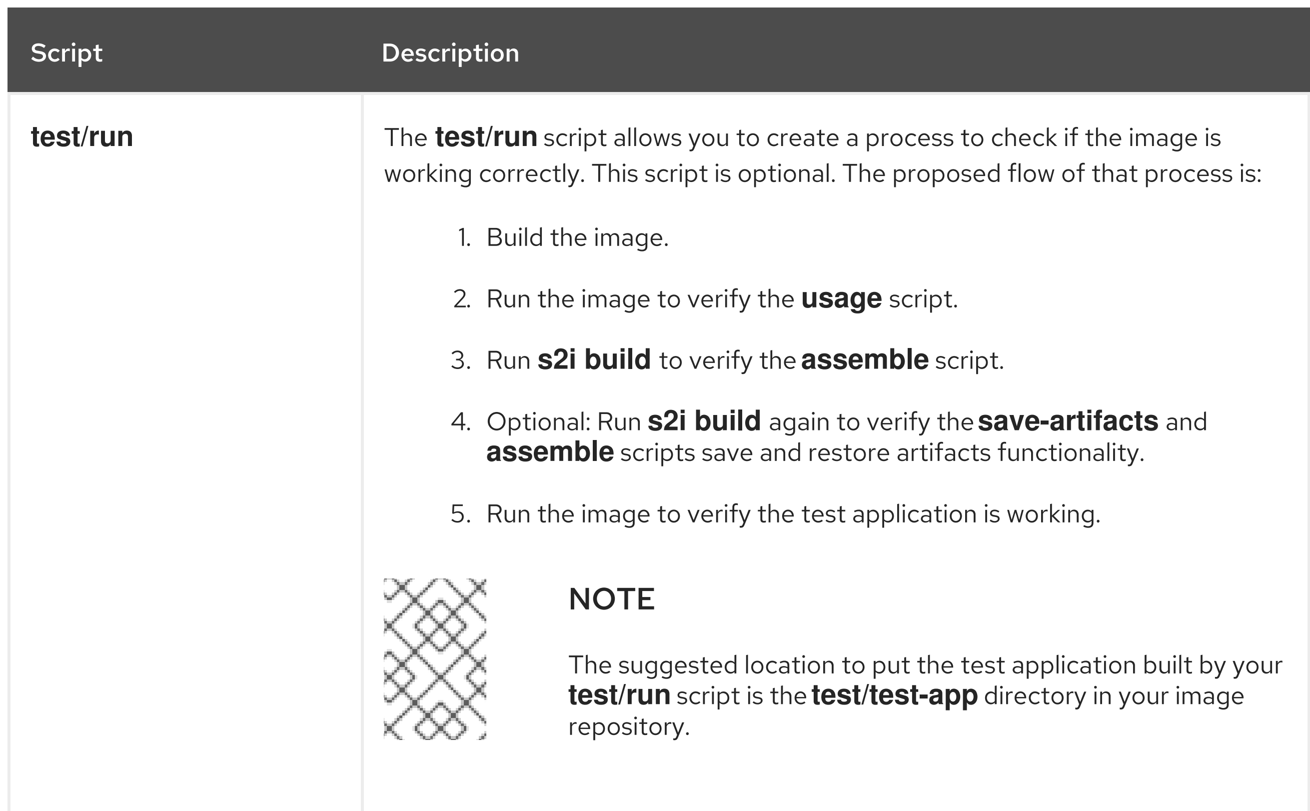
- **image:///path_to_scripts_dir**: absolute path inside the image to a directory where the S2I scripts are located.

- **file:///path_to_scripts_dir**: relative or absolute path to a directory on the host where the S2I scripts are located.

- **http(s)://path_to_scripts_dir**: URL to a directory where the S2I scripts are located.

Table 4.2. S2I scripts

Script	Description
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: 1. Optional: Restore build artifacts. If you want to support incremental builds, make sure to define **save-artifacts** as well. 2. Place the application source in the desired location. 3. Build the application artifacts. 4. Install the artifacts into locations appropriate for them to run.
run	The **run** script executes your application. This script is required.
save-artifacts	The **save-artifacts** script gathers all dependencies that can speed up the build processes that follow. This script is optional. For example: • For Ruby, **gems** installed by Bundler. • For Java, **.m2** contents. 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.



Script	Description
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: 1. Build the image. 2. Run the image to verify the **usage** script. 3. Run **s2i build** to verify the **assemble** script. 4. Optional: Run **s2i build** again to verify the **save-artifacts** and **assemble** scripts save and restore artifacts functionality. 5. Run the image to verify the test application is working. NOTE 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

As an Source-to-Image (S2I) builder image author, you can test your S2I image locally and use the Red Hat OpenShift Service on AWS 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 Red Hat OpenShift Service on AWS 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.

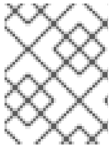
S2I combines the application source code and builder image, so to test it you need a sample application source to verify that the source successfully transforms into a runnable container image. The sample application should be simple, but it should exercise the crucial steps of **assemble** and **run** 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 Red Hat OpenShift Service on AWS 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 Red Hat OpenShift Service on AWS to build and push the image. Define a Docker build that points to your repository.

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

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

CHAPTER 5. MANAGING IMAGES

5.1. MANAGING IMAGES OVERVIEW

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

5.1.1. Images overview

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

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

5.2. TAGGING IMAGES

The following sections provide an overview and instructions for using image tags in the context of container images for working with Red Hat OpenShift Service on AWS image streams and their tags.

5.2.1. Image tags

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

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

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

Red Hat OpenShift Service on AWS provides the **oc tag** command, which is similar to the **docker tag** command, but operates on image streams instead of directly on images.

5.2.2. Image tag conventions

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

If there is too much information embedded in a tag name, like **v2.0.1-may-2019**, the tag points to just one revision of an image and is never updated. Using default image pruning options, such an image is never removed.

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

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

Table 5.1. Image tag naming conventions

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

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

5.2.3. Adding tags to image streams

An image stream in Red Hat OpenShift Service on AWS comprises zero or more container images identified by tags.

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

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

Procedure

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

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

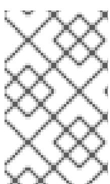
  For example, to configure the **ruby** image stream **static-2.0** tag to always refer to the current image for the **ruby** image stream **2.0** tag:

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

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

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

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



NOTE

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

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

5.2.4. Removing tags from image streams

You can remove tags from an image stream.

Procedure

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

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

  or:

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

5.2.5. Referencing images in imagestreams

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

Table 5.2. Imagestream reference types

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

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

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

image streams.

Procedure

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

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

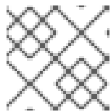
- To reference an image for a given image stream and image **sha** ID, use **ImageStreamImage**:

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

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

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

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



NOTE

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

You can also reference a third-party registry:

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

Or an image with a digest:

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

5.3. IMAGE PULL POLICY

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

5.3.1. Image pull policy overview

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

Table 5.3. imagePullPolicy values

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

Value	Description
Never	Never pull the image.

If a container **imagePullPolicy** parameter is not specified, Red Hat OpenShift Service on AWS sets it based on the image tag:

1. If the tag is **latest**, Red Hat OpenShift Service on AWS defaults **imagePullPolicy** to **Always**.
2. Otherwise, Red Hat OpenShift Service on AWS defaults **imagePullPolicy** to **IfNotPresent**.

5.4. USING IMAGE PULL SECRETS

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 and no additional action should be required.

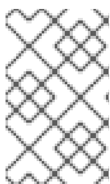
However, for other scenarios, such as referencing images across Red Hat OpenShift Service on AWS projects or from secured registries, additional configuration steps are required.

You can obtain the image pull secret from Red Hat OpenShift Cluster Manager . This pull secret is called **pullSecret**.

You use this pull secret to authenticate with the services that are provided by the included authorities, Quay.io and registry.redhat.io, which serve the container images for Red Hat OpenShift Service on AWS components.

5.4.1. Allowing pods to reference images across projects

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



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

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

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

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

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

5.4.2. Allowing pods to reference images from other secured registries

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

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

The Docker credentials file and the associated pull secret can contain multiple references to the same registry, each with its own set of credentials.

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
```

Procedure

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

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

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

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

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

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

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

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

5.4.2.1. Pulling from private registries with delegated authentication

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

Procedure

1. Create a secret for the delegated authentication server:

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

secret/redhat-connect-sso
```

2. Create a secret for the private registry:

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

```
    private-registry

secret/private-registry
```

CHAPTER 6. MANAGING IMAGE STREAMS

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

6.1. WHY USE IMAGESTREAMS

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

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

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

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

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

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

- Red Hat OpenShift Service on AWS's integrated registry.
- An external registry, for example registry.redhat.io or quay.io.
- Other image streams in the Red Hat OpenShift Service on AWS 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, Red Hat OpenShift Service on AWS 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, Red Hat OpenShift Service on AWS 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

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 (1)
  namespace: test
spec: {}
status:
  dockerImageRepository: 172.30.56.218:5000/test/origin-ruby-sample (2)
  tags:
  - items:
    - created: 2017-09-02T10:15:09Z
      dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d (3)
      generation: 2
      image: sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab3fc5 (4)
    - 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 (5)
```

1. The name of the image stream.
2. Docker repository path where new images can be pushed to add or update them in this image stream.
3. The SHA identifier that this image stream tag currently references. Resources that reference this image stream tag use this identifier.
4. The SHA identifier that this image stream tag previously referenced. Can be used to rollback to an older image.
5. The image stream tag name.

6.3. IMAGE STREAM IMAGES

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

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

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

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

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

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

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

6.4. IMAGE STREAM TAGS

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

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

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

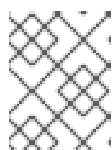
Image stream tag with two images in its history

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



NOTE

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

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

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

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

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

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

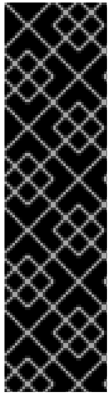
6.5. IMAGE STREAM CHANGE TRIGGERS

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

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

6.6. WORKING WITH IMAGE STREAMS

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



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.

6.6.1. Getting information about image streams

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

Procedure

- 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:

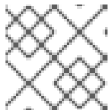
```
$ oc describe istag/<image-stream>:<tag-name>
```

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
```

6.6.2. Adding tags to an image stream

You can add additional tags to image streams.

Procedure

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

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

For example:

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

Example output

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

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

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

Example output

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

latest
  tagged from
python@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25

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

3.5
  tagged from centos/python-35-centos7

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

6.6.3. Adding tags for an external image

You can add tags for external images.

Procedure

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

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

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

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

Example output

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

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

6.6.4. Updating image stream tags

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

Procedure

- Update a tag:

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

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

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

Example output

```
Tag python:latest set to
python@sha256:438208801c4806548460b27bd1fbcb7bb188273d13871ab43f.
```

6.6.5. Removing image stream tags

You can remove old tags from an image stream.

Procedure

- Remove old tags from an image stream:

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

For example:

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

Example output

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

See Removing deprecated image stream tags from the Cluster Samples Operator for more information on how the Cluster Samples Operator handles deprecated image stream tags.

6.6.6. Configuring periodic importing of image stream tags

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

Procedure

1. Schedule importing images:

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

   For example:

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

   Example output

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

   This command causes Red Hat OpenShift Service on AWS 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.

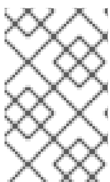
```
$ oc tag <repositiory/image> <image-name:tag>
```

6.7. IMPORTING AND WORKING WITH IMAGES AND IMAGE STREAMS

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

6.7.1. Importing images and image streams from private registries

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



NOTE

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

Procedure

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

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

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

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

During the import process, Red Hat OpenShift Service on AWS picks up the secrets and provides them to the remote party.

6.7.1.1. Allowing pods to reference images from other secured registries

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

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

The Docker credentials file and the associated pull secret can contain multiple references to the same registry, each with its own set of credentials.

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
```

Procedure

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

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

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

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

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

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

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

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

6.7.2. Working with manifest lists

You can import a single sub-manifest, or all manifests, of a manifest list when using **oc import-image** or **oc tag** CLI commands by adding the **--import-mode** flag.

Refer to the commands below to create an image stream that includes a single sub-manifest or multi-architecture images.

Procedure

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

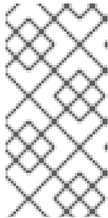
```
$ oc import-image <multiarch-image-stream-tag>  --from=
<registry>/<project_name>/<image-name> \
--import-mode='PreserveOriginal' --reference-policy=local --confirm
```

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**.

Limitations

Working with manifest lists has the following limitations:

- 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.

6.7.2.1. Configuring periodic importing of manifest lists

To periodically re-import a manifest list, you can use the **--scheduled** flag.

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
```

6.7.2.2. Configuring SSL/TSL when importing manifest lists

To configure SSL/TSL when importing a manifest list, you can use the **--insecure** flag.

Procedure

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

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

6.7.3. Specifying architecture for --import-mode

You can swap your imported image stream between multi-architecture and single architecture by excluding or including the **--import-mode=** flag

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'
```

6.7.4. Configuration fields for --import-mode

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

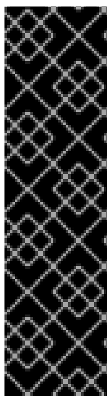
Parameter	Description
Legacy	The default option for **--import-mode**. When specified, the manifest list is discarded, and a single sub-manifest is imported. The platform is chosen in the following order of priority: 1. Tag annotations 2. Control plane architecture 3. Linux/AMD64 4. The first manifest 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

Image streams, being Red Hat OpenShift Service on AWS native resources, work with all native resources available in Red Hat OpenShift Service on AWS, such as **Build** or **DeploymentConfigs** resources. It is also possible to make them work with native Kubernetes resources, such as **Job**, **ReplicationController**, **ReplicaSet** or Kubernetes **Deployment** resources.

7.1. ENABLING IMAGE STREAMS WITH KUBERNETES RESOURCES

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



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.

Procedure

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

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

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

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

Imagestream with image lookup enabled

```
apiVersion: 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
```

You can enable image lookup on a specific resource.

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

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

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

Deployment with image lookup enabled

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

You can disable image lookup.

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

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

CHAPTER 8. TRIGGERING UPDATES ON IMAGE STREAM CHANGES

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

8.1. RED HAT OPENSHIFT SERVICE ON AWS RESOURCES

Red Hat OpenShift Service on AWS deployment configurations and build configurations can be automatically triggered by changes to image stream tags. The triggered action can be run using the new value of the image referenced by the updated image stream tag.

8.2. TRIGGERING KUBERNETES RESOURCES

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

The annotation is defined as follows:

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    image.openshift.io/triggers:
      [
       {
         "from": {
           "kind": "ImageStreamTag", (1)
           "name": "example:latest", (2)
           "namespace": "myapp" (3)
         },
         "fieldPath": "spec.template.spec.containers[?(@.name==\"web\")].image", (4)
         "paused": false (5)
       },
      # ...
      ]
# ...
```

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

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

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

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

8.3. SETTING THE IMAGE TRIGGER ON KUBERNETES RESOURCES

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

Procedure

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

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

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"}]'
# ...
```

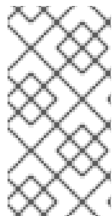
Unless the deployment is paused, this pod template update automatically causes a deployment to occur with the new image value.

CHAPTER 9. IMAGE CONFIGURATION RESOURCES

Use the following procedure to configure image registries.

9.1. IMAGE CONTROLLER CONFIGURATION PARAMETERS

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



NOTE

Parameters such as **DisableScheduledImport**, **MaxImagesBulkImportedPerRepository**, **MaxScheduledImportsPerMinute**, **ScheduledImageImportMinimumIntervalSeconds**, **InternalRegistryHostname** are not configurable.

Parameter	Description
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. Every element of this list contains a location of the registry specified by the registry domain name. **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. **insecure**: Insecure indicates whether the registry is secure or insecure. By default, if not otherwise specified, the registry is assumed to be secure.
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. 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.

Parameter	Description
registrySources	Contains configuration that determines how the container runtime should treat individual registries when accessing images for builds and pods. For instance, whether or not to allow insecure access. It does not contain configuration for the internal cluster registry. **insecureRegistries**: Registries which 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**. **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. **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. **containerRuntimeSearchRegistries**: Registries for which image pull and push actions are allowed using image short names. All other registries are blocked. Either **blockedRegistries** or **allowedRegistries** can be set, but not both.

WARNING

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

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

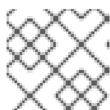
Parameter	Description

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. CONFIGURING IMAGE REGISTRY SETTINGS

You can configure image registry settings by editing the **image.config.openshift.io/cluster** custom resource (CR). When changes to the registry are applied to the **image.config.openshift.io/cluster** CR, the Machine Config Operator (MCO) performs the following sequential actions:

1. Cordons the node
2. Applies changes by restarting CRI-O
3. Uncordons the node



NOTE

The MCO does not restart nodes when it detects changes.

Procedure

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

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

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

```
apiVersion: config.openshift.io/v1
kind: Image (1)
metadata:
  annotations:
    release.openshift.io/create-only: "true"
  creationTimestamp: "2019-05-17T13:44:26Z"
  generation: 1
  name: cluster
  resourceVersion: "8302"
  selfLink: /apis/config.openshift.io/v1/images/cluster
  uid: e34555da-78a9-11e9-b92b-06d6c7da38dc
spec:
  allowedRegistriesForImport: (2)
    - domainName: quay.io
      insecure: false
```

```
  additionalTrustedCA: (3)
    name: myconfigmap
  registrySources: (4)
    allowedRegistries:
    - example.com
    - quay.io
    - registry.redhat.io
    - image-registry.openshift-image-registry.svc:5000
    - reg1.io/myrepo/myapp:latest
    insecureRegistries:
    - insecure.com
status:
  internalRegistryHostname: image-registry.openshift-image-registry.svc:5000
```

1. **Image**: Holds cluster-wide information about how to handle images. The canonical, and only valid name is **cluster**.

2. **allowedRegistriesForImport**: Limits the container image registries from which normal users may import images. Set this list to the registries that you trust to contain valid images, and that you want applications to be able to import from. Users with permission to create images or **ImageStreamMappings** from the API are not affected by this policy. Typically only cluster administrators have the appropriate permissions.

3. **additionalTrustedCA**: A reference to a config map containing additional certificate authorities (CA) that are trusted during image stream import, pod image pull, **openshift-image-registry** pullthrough, and builds. The namespace for this config map is **openshift-config**. The format of the config map is to use the registry hostname as the key, and the PEM certificate as the value, for each additional registry CA to trust.

4. **registrySources**: Contains configuration that determines whether the container runtime allows or blocks individual registries when accessing images for builds and pods. Either the **allowedRegistries** parameter or the **blockedRegistries** parameter can be set, but not both. You can also define whether or not to allow access to insecure registries or registries that allow registries that use image short names. This example uses the **allowedRegistries** parameter, which defines the registries that are allowed to be used. The insecure registry **insecure.com** is also allowed. The **registrySources** parameter does not contain configuration for the internal cluster registry.

NOTE

When the **allowedRegistries** parameter is defined, all registries, including the registry.redhat.io and quay.io registries and the default OpenShift image registry, are blocked unless explicitly listed. If you use the parameter, to prevent pod failure, you must add the **registry.redhat.io** and **quay.io** registries and the **internalRegistryHostname** to the **allowedRegistries** list, as they are required by payload images within your environment. Do not add the **registry.redhat.io** and **quay.io** registries to the **blockedRegistries** list.

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

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

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

```
$ oc get nodes
```

Example output

```
NAME                                         STATUS                     ROLES                   AGE   VERSION
ip-10-0-137-182.us-east-2.compute.internal   Ready,SchedulingDisabled   worker
65m   v1.28.5
ip-10-0-139-120.us-east-2.compute.internal   Ready,SchedulingDisabled   control-plane
74m   v1.28.5
ip-10-0-176-102.us-east-2.compute.internal   Ready                      control-plane          75m
v1.28.5
ip-10-0-188-96.us-east-2.compute.internal    Ready                      worker                 65m
v1.28.5
ip-10-0-200-59.us-east-2.compute.internal    Ready                      worker                 63m
v1.28.5
ip-10-0-223-123.us-east-2.compute.internal   Ready                      control-plane          73m
v1.28.5
```

9.2.1. Adding specific registries

You can add a list of registries, and optionally an individual repository within a registry, that are permitted for image pull and push actions by editing the **image.config.openshift.io/cluster** custom resource (CR). Red Hat OpenShift Service on AWS applies the changes to this CR to all nodes in the cluster.

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

WARNING

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

Procedure

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

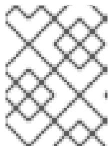
```
$ oc edit image.config.openshift.io/cluster
```

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

```
apiVersion: config.openshift.io/v1
kind: Image
metadata:
  annotations:
    release.openshift.io/create-only: "true"
  creationTimestamp: "2019-05-17T13:44:26Z"
  generation: 1
  name: cluster
  resourceVersion: "8302"
  selfLink: /apis/config.openshift.io/v1/images/cluster
  uid: e34555da-78a9-11e9-b92b-06d6c7da38dc
spec:
  registrySources: (1)
    allowedRegistries: (2)
    - example.com
    - quay.io
    - registry.redhat.io
    - reg1.io/myrepo/myapp:latest
    - image-registry.openshift-image-registry.svc:5000
status:
  internalRegistryHostname: image-registry.openshift-image-registry.svc:5000
```

1. Contains configurations that determine how the container runtime should treat individual registries when accessing images for builds and pods. It does not contain configuration for the internal cluster registry.

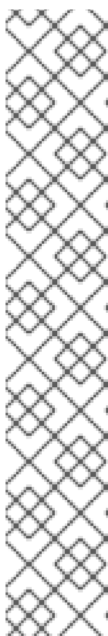
2. Specify registries, and optionally a repository in that registry, to use for image pull and push actions. All other registries are blocked.



NOTE

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

The Machine Config Operator (MCO) watches the **image.config.openshift.io/cluster** resource for any changes to the registries. When the MCO detects a change, it drains the nodes, applies the change, and uncordons the nodes. After the nodes return to the **Ready** state, the allowed registries list is used to update the image signature policy in the **/etc/containers/policy.json** file on each node.



NOTE

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

For example:

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

9.2.2. Blocking specific registries

You can block any registry, and optionally an individual repository within a registry, by editing the **image.config.openshift.io/cluster** custom resource (CR). Red Hat OpenShift Service on AWS applies the changes to this CR to all nodes in the cluster.

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

WARNING

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

Procedure

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

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

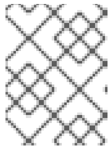
  The following is an example **image.config.openshift.io/cluster** CR with a blocked list:

  ```
  apiVersion: config.openshift.io/v1
  kind: Image
  metadata:
    annotations:
      release.openshift.io/create-only: "true"
    creationTimestamp: "2019-05-17T13:44:26Z"
    generation: 1
  ```

```
  name: cluster
  resourceVersion: "8302"
  selfLink: /apis/config.openshift.io/v1/images/cluster
  uid: e34555da-78a9-11e9-b92b-06d6c7da38dc
spec:
  registrySources: 1
    blockedRegistries: 2
    - untrusted.com
    - reg1.io/myrepo/myapp:latest
status:
  internalRegistryHostname: image-registry.openshift-image-registry.svc:5000
```

1. Contains configurations that determine how the container runtime should treat individual registries when accessing images for builds and pods. It does not contain configuration for the internal cluster registry.

2. Specify registries, and optionally a repository in that registry, that should not be used for image pull and push actions. All other registries are allowed.



NOTE

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

The Machine Config Operator (MCO) watches the **image.config.openshift.io/cluster** resource for any changes to the registries. When the MCO detects a change, it drains the nodes, applies the change, and uncordons the nodes. After the nodes return to the **Ready** state, changes to the blocked registries appear in the **/etc/containers/registries.conf** file on each node.

9.2.3. Allowing insecure registries

You can add insecure registries, and optionally an individual repository within a registry, by editing the **image.config.openshift.io/cluster** custom resource (CR). Red Hat OpenShift Service on AWS applies the changes to this CR to all nodes in the cluster.

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

WARNING

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

Procedure

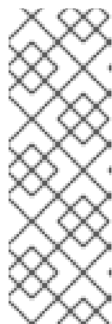
- Edit the **image.config.openshift.io/cluster** custom resource:

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

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

```
apiVersion: config.openshift.io/v1
kind: Image
metadata:
  annotations:
    release.openshift.io/create-only: "true"
  creationTimestamp: "2019-05-17T13:44:26Z"
  generation: 1
  name: cluster
  resourceVersion: "8302"
  selfLink: /apis/config.openshift.io/v1/images/cluster
  uid: e34555da-78a9-11e9-b92b-06d6c7da38dc
spec:
  registrySources: 1
    insecureRegistries: 2
    - insecure.com
    - reg4.io/myrepo/myapp:latest
    allowedRegistries:
    - example.com
    - quay.io
    - registry.redhat.io
    - insecure.com 3
    - reg4.io/myrepo/myapp:latest
    - image-registry.openshift-image-registry.svc:5000
status:
  internalRegistryHostname: image-registry.openshift-image-registry.svc:5000
```

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



NOTE

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

The Machine Config Operator (MCO) watches the **image.config.openshift.io/cluster** CR for any changes to the registries, then drains and uncordons the nodes when it detects changes. After the nodes return to the **Ready** state, changes to the insecure and blocked registries appear in the **/etc/containers/registries.conf** file on each node.

9.2.4. 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). Red Hat OpenShift Service on AWS applies the changes to this CR to all nodes in the cluster.

An image short name enables you to search for images without including the fully qualified domain name in the pull spec. For example, you could use **rhel7/etcd** instead of **registry.access.redhat.com/rhe7/etcd**.

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 frequently, 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.

WARNING

Using image short names with public registries is strongly discouraged because the image might not deploy if the public registry requires authentication. Use fully-qualified image names with public registries.

Red Hat internal or private registries typically support the use of image short names.

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.

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.

The Machine Config Operator (MCO) watches the **image.config.openshift.io/cluster** resource for any changes to the registries. When the MCO detects a change, it drains the nodes, applies the change, and uncordons the nodes. 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.

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.

Procedure

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

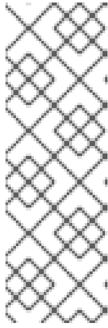
```
$ oc edit image.config.openshift.io/cluster
```

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: 1
    - reg1.io
    - reg2.io
    - reg3.io
    allowedRegistries: 2
    - 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. Specify registries to use with image short names. You should use image short names with only internal or private registries to reduce possible security risks.

2. Ensure that any registries listed under **containerRuntimeSearchRegistries** are included in the **allowedRegistries** list.



NOTE

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

9.2.5. Configuring additional trust stores for image registry access

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

Prerequisites

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

Procedure

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

The config map key is the hostname of a registry with the port for which this CA is to be trusted, and the PEM certificate content is the value, for each additional registry CA to trust.

Image registry CA config map example

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

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

You can configure additional CAs with the following procedure.

- To configure an additional CA:

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

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

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

9.3. UNDERSTANDING IMAGE REGISTRY REPOSITORY MIRRORING

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

- Configure your Red Hat OpenShift Service on AWS 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.

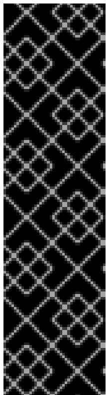
Repository mirroring in Red Hat OpenShift Service on AWS includes the following attributes:

- 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 Red Hat OpenShift Service on AWS cluster.
- When a node makes a request for an image from the source repository, it tries each mirrored repository in turn until it finds the requested content. If all mirrors fail, the cluster tries the source repository. If successful, the image is pulled to the node.

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

- At Red Hat OpenShift Service on AWS installation:
  By pulling container images needed by Red Hat OpenShift Service on AWS and then bringing those images behind your company’s firewall, you can install Red Hat OpenShift Service on AWS into a datacenter that is in a disconnected environment.
- After Red Hat OpenShift Service on AWS installation:
  If you did not configure mirroring during Red Hat OpenShift Service on AWS 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

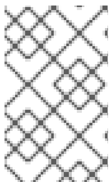
Using an **ImageContentSourcePolicy** (ICSP) object to configure repository mirroring is a deprecated feature. Deprecated functionality is still included in Red Hat OpenShift Service on AWS and continues to be supported; however, it will be removed in a future release of this product and is not recommended 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 an **ImageDigestMirrorSet** YAML file. For more information, see "Converting ImageContentSourcePolicy (ICSP) files for image registry repository mirroring" in the following section.

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 requested from the source repository.

For new clusters, you can use IDMS, ITMS, and ICSP CRs objects as desired. 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 using 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.

9.3.1. 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 **dedicated-admin** role.

Procedure

1. Configure mirrored repositories, by either:
   - Setting up a mirrored repository with Red Hat Quay, as described in Red Hat Quay Repository Mirroring. Using Red Hat Quay allows you to copy images from one repository to another and also automatically sync those repositories repeatedly over time.

- Using a tool such as **skopeo** to copy images manually from the source repository to the mirrored repository.
  For example, after installing the skopeo RPM package on a Red Hat Enterprise Linux (RHEL) 7 or RHEL 8 system, use the **skopeo** command as shown in this example:

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

  In this example, you have a container image registry that is named **example.io** with an image repository named **example** to which you want to copy the **ubi9/ubi-minimal** image from **registry.access.redhat.com**. After you create the mirrored registry, you can configure your Red Hat OpenShift Service on AWS cluster to redirect requests made of the source repository to the mirrored repository.

2. Log in to your Red Hat OpenShift Service on AWS cluster.

3. Create a postinstallation mirror configuration 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 (1)
kind: ImageDigestMirrorSet (2)
metadata:
  name: ubi9repo
spec:
  imageDigestMirrors: (3)
  - mirrors:
    - example.io/example/ubi-minimal (4)
    - example.com/example/ubi-minimal (5)
    source: registry.access.redhat.com/ubi9/ubi-minimal (6)
    mirrorSourcePolicy: AllowContactingSource (7)
  - mirrors:
    - mirror.example.com/redhat
    source: registry.example.com/redhat (8)
    mirrorSourcePolicy: AllowContactingSource
  - mirrors:
    - mirror.example.com
    source: registry.example.com (9)
    mirrorSourcePolicy: AllowContactingSource
  - mirrors:
    - mirror.example.net/image
    source: registry.example.com/example/myimage (10)
    mirrorSourcePolicy: AllowContactingSource
  - mirrors:
    - mirror.example.net
    source: registry.example.com/example (11)
    mirrorSourcePolicy: AllowContactingSource
  - mirrors:
    - mirror.example.net/registry-example-com
    source: registry.example.com (12)
    mirrorSourcePolicy: AllowContactingSource
```

1. Indicates the API to use with this CR. This must be **config.openshift.io/v1**.
2. Indicates the kind of object according to the pull type:
   - **ImageDigestMirrorSet**: Pulls a digest reference image.
   - **ImageTagMirrorSet**: Pulls a tag reference image.
3. Indicates the type of image pull method, either:
   - **imageDigestMirrors**: Use for an **ImageDigestMirrorSet** CR.
   - **imageTagMirrors**: Use for an **ImageTagMirrorSet** CR.
4. Indicates the name of the mirrored image registry and repository.
5. Optional: Indicates a secondary mirror repository for each target repository. If one mirror is down, the target repository can use another mirror.
6. Indicates the registry and repository source, which is the repository that is referred to in image pull specifications.
7. Optional: Indicates the fallback policy if the image pull fails:
   - **AllowContactingSource**: Allows continued attempts to pull the image from the source repository. This is the default.
   - **NeverContactSource**: Prevents continued attempts to pull the image from the source repository.
8. Optional: Indicates a namespace inside a registry, which allows you to use any image in that namespace. If you use a registry domain as a source, the object is applied to all repositories from the registry.
9. Optional: Indicates a registry, which allows you to use any image in that registry. If you specify a registry name, the object is applied to all repositories from a source registry to a mirror registry.
10. Pulls the image **registry.example.com/example/myimage@sha256:...** from the mirror **mirror.example.net/image@sha256:..**.
11. Pulls the image **registry.example.com/example/image@sha256:...** in the source registry namespace from the mirror **mirror.example.net/image@sha256:...**.
12. Pulls the image **registry.example.com/myimage@sha256** from the mirror registry **example.net/registry-example-com/myimage@sha256:...**.

- 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 (1)
  source: quay.io/openshift-release-dev/ocp-release (2)
- mirrors:
  - mirror.registry.com:443/ocp/release
  source: quay.io/openshift-release-dev/ocp-v4.0-art-dev
```

1. Specifies the name of the mirror image registry and repository.
2. Specifies the online registry and repository containing the content that is mirrored.

4. Create the new object:

```
$ 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.

5. To check that the mirrored configuration settings are applied, do the following on one of the nodes.

a. List your nodes:

```
$ oc get node
```

Example output

```
NAME                           STATUS                    ROLES    AGE  VERSION
ip-10-0-137-44.ec2.internal    Ready                     worker   7m   v1.28.5
ip-10-0-138-148.ec2.internal   Ready                     master   11m  v1.28.5
ip-10-0-139-122.ec2.internal   Ready                     master   11m  v1.28.5
ip-10-0-147-35.ec2.internal    Ready                     worker   7m   v1.28.5
ip-10-0-153-12.ec2.internal    Ready                     worker   7m   v1.28.5
ip-10-0-154-10.ec2.internal    Ready                     master   11m  v1.28.5
```

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 were applied. The final two entries are marked **digest-only** and **tag-only** respectively.

Example output

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

[[registry]]
  prefix = ""
  location = "registry.access.redhat.com/ubi9/ubi-minimal" (1)

  [[registry.mirror]]
    location = "example.io/example/ubi-minimal" (2)
    pull-from-mirror = "digest-only" (3)

  [[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 (4)

  [[registry.mirror]]
    location = "example.io/example/ubi-minimal-tag"
    pull-from-mirror = "tag-only" (5)
```

1. Indicates the repository that is referred to in a pull spec.
2. Indicates the mirror for that repository.
3. Indicates that the image pull from the mirror is a digest reference image.
4. Indicates that the **NeverContactSource** parameter is set for this repository.
5. 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 repository mirroring

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

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

9.3.2. 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 Red Hat OpenShift Service on AWS 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 **dedicated-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.

CHAPTER 10. USING TEMPLATES

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

10.1. UNDERSTANDING TEMPLATES

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

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

10.2. UPLOADING A TEMPLATE

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

Procedure

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

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

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

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

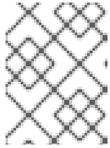
The template is now available for selection using the web console or the CLI.

10.3. CREATING AN APPLICATION BY USING THE WEB CONSOLE

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

Procedure

1. Select **Developer** from the context selector at the top of the web console navigation menu.
2. While in the desired project, click **+Add**
3. Click **All services** in the **Developer Catalog** tile.
4. Click **Builder Images** under **Type** to see the available builder images.



NOTE

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

```
kind: "ImageStream"
apiVersion: "v1"
metadata:
  name: "ruby"
  creationTimestamp: null
spec:
# ...
  tags:
    - name: "2.6"
      annotations:
        description: "Build and run Ruby 2.6 applications"
        iconClass: "icon-ruby"
        tags: "builder,ruby" (1)
        supports: "ruby:2.6,ruby"
        version: "2.6"
# ...
```

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

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

10.4. CREATING OBJECTS FROM TEMPLATES BY USING THE CLI

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

10.4.1. Adding labels

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

Procedure

- Add labels in the template from the command line:

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

10.4.2. Listing parameters

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

Procedure

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

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

Alternatively, if the template is already uploaded:

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

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

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

Example output

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

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

10.4.3. Generating a list of objects

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

Procedure

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

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

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

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

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

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

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

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

3. You can override any parameter values defined in the file by adding the **-p** option for each **<name>=<value>** pair you want to override. A parameter reference appears in any text field inside the template items.
For example, in the following the **POSTGRESQL_USER** and **POSTGRESQL_DATABASE** parameters of a template are overridden to output a configuration with customized environment variables:

    a. Creating a List of objects from a template

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

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

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

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

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

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

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

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

10.5. MODIFYING UPLOADED TEMPLATES

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

Procedure

- Modify a template that has already been uploaded:

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

10.6. WRITING TEMPLATES

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

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

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

10.6.1. Writing the template description

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

the name of the language the template is related to for example, Java, PHP, Ruby, and so on.

The following is an example of template description metadata:

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


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

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

   Example 10.1. Available icons

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

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

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

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

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

7. The name of the person or organization providing the template.

8. A URL referencing further documentation for the template.

9. A URL where support can be obtained for the template.

10. An instructional message that is displayed when this template is instantiated. This field should inform the user how to use the newly created resources. Parameter substitution is performed on the message before being displayed so that generated credentials and other parameters can be included in the output. Include links to any next-steps documentation that users should follow.

10.6.2. Writing template labels

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

The following is an example of template object labels:

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

1. A label that is applied to all objects created from this template.

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

10.6.3. Writing template parameters

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

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

When using the **${PARAMETER_NAME}** syntax, multiple parameter references can be combined in a single field and the reference can be embedded within fixed data, such as **"http://${PARAMETER_1}${PARAMETER_2}"**. Both parameter values are substituted and the resulting value is a quoted string.

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

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

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

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

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

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

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

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

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

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

NOTE

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

Example YAML template with a modifier

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

Example JSON template with a modifier

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

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

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

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

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

10.6.4. Writing the template object list

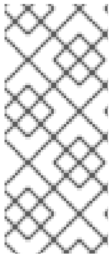
The main portion of the template is the list of objects which is created when the template is instantiated. This can be any valid API object, such as a build configuration, deployment configuration, or service. The object is created exactly as defined here, with any parameter values substituted in prior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

The following is an example of an object list:

```
kind: "Template"
```

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

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



NOTE

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

10.6.5. Marking a template as bindable

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

Procedure

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

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

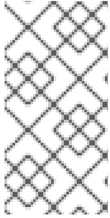
10.6.6. Exposing template object fields

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

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

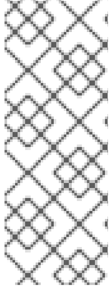
Each annotation key, with its prefix removed, is passed through to become a key in a **bind** response.

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



NOTE

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



NOTE

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

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

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

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

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

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

Procedure

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

10.6.7. Waiting for template readiness

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

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

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

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

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

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

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

The following is an example template extract, which uses the **wait-for-ready** annotation. Further examples can be found in the Red Hat OpenShift Service on AWS quick start templates.

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

Additional recommendations

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

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

10.6.8. Creating a template from existing objects

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

Procedure

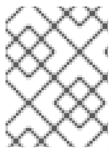
- Export objects in a project in YAML form:

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

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

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

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



NOTE

Using the **all** alias is not recommended because the contents might vary across different clusters and versions. Instead, specify all required resources.

CHAPTER 11. USING RUBY ON RAILS

Ruby on Rails is a web framework written in Ruby. This guide covers using Rails 4 on Red Hat OpenShift Service on AWS.

WARNING

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

11.1. PREREQUISITES

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

11.2. SETTING UP THE DATABASE

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

Procedure

1. Install the database:

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

2. Initialize the database:

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

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

3. Start the database:

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

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

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

Note that the user created has no password.

11.3. WRITING YOUR APPLICATION

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

Procedure

1. Install the Rails gem:

   ```
   $ gem install rails
   ```

   Example output

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

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

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

3. Change into your new application directory:

   ```
   $ cd rails-app
   ```

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

   ```
   gem 'pg'
   ```

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

   ```
   $ bundle install
   ```

6. In addition to using the **postgresql** database with the **pg** gem, you also must ensure that the **config/database.yml** is using the **postgresql** adapter.
   Make sure you updated **default** section in the **config/database.yml** file, so it looks like this:

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

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

```
$ rake db:create
```

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

11.3.1. Creating a welcome page

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

To have a custom welcome page must do following steps:

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

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

Procedure

1. Run Rails generator:

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

All the necessary files are created.

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

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

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

```
$ rails server
```

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

11.3.2. Configuring application for Red Hat OpenShift Service on AWS

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

Procedure

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

Sample config/database YAML file

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

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

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

11.3.3. Storing your application in Git

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

Prerequisites

- Install git.

Procedure

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

```
$ ls -1
```

Example output

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

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

```
$ git init
```

```
$ git add .
```

```
$ git commit -m "initial commit"
```

After your application is committed you must push it to a remote repository. GitHub account, in which you create a new repository.

3. Set the remote that points to your **git** repository:

```
$ git remote add origin git@github.com:<namespace/repository-name>.git
```

4. Push your application to your remote git repository.

```
$ git push
```

11.4. DEPLOYING YOUR APPLICATION TO RED HAT OPENSHIFT SERVICE ON AWS

You can deploy you application to Red Hat OpenShift Service on AWS.

After creating the **rails-app** project, you are automatically switched to the new project namespace.

Deploying your application in Red Hat OpenShift Service on AWS involves three steps:

- Creating a database service from Red Hat OpenShift Service on AWS's PostgreSQL image.
- Creating a frontend service from Red Hat OpenShift Service on AWS's Ruby 2.0 builder image and your Ruby on Rails source code, which are wired with the database service.
- Creating a route for your application.

11.4.1. Creating the database service

Procedure

Your Rails application expects a running database service. For this service use PostgreSQL database image.

To create the database service, use the **oc new-app** command. To this command you must pass some necessary environment variables which are used inside the database container. These environment variables are required to set the username, password, and name of the database. You can change the values of these environment variables to anything you would like. The variables are as follows:

- POSTGRESQL_DATABASE
- POSTGRESQL_USER
- POSTGRESQL_PASSWORD

Setting these variables ensures:

- A database exists with the specified name.
- A user exists with the specified name.
- The user can access the specified database with the specified password.

Procedure

1. Create the database service:

```
$ oc new-app postgresql -e POSTGRESQL_DATABASE=db_name -e
POSTGRESQL_USER=username -e POSTGRESQL_PASSWORD=password
```

   To also set the password for the database administrator, append to the previous command with:

```
-e POSTGRESQL_ADMIN_PASSWORD=admin_pw
```

2. Watch the progress:

```
$ oc get pods --watch
```

11.4.2. Creating the frontend service

To bring your application to Red Hat OpenShift Service on AWS, you must specify a repository in which your application lives.

Procedure

1. Create the frontend service and specify database related environment variables that were setup when creating the database service:

```
$ oc new-app path/to/source/code --name=rails-app -e POSTGRESQL_USER=username -e
POSTGRESQL_PASSWORD=password -e POSTGRESQL_DATABASE=db_name -e
DATABASE_SERVICE_NAME=postgresql
```

   With this command, Red Hat OpenShift Service on AWS fetches the source code, sets up the builder, builds your application image, and deploys the newly created image together with the specified environment variables. The application is named **rails-app**.

2. Verify the environment variables have been added by viewing the JSON document of the **rails-app** deployment config:

```
$ oc get dc rails-app -o json
```

   You should see the following section:

   Example output

```
env": [
    {
        "name": "POSTGRESQL_USER",
        "value": "username"
    },
    {
        "name": "POSTGRESQL_PASSWORD",
        "value": "password"
    },
    {
        "name": "POSTGRESQL_DATABASE",
        "value": "db_name"
```

```
        },
        {
            "name": "DATABASE_SERVICE_NAME",
            "value": "postgresql"
        }

    ],
```

3. Check the build process:

```
$ oc logs -f build/rails-app-1
```

4. After the build is complete, look at the running pods in Red Hat OpenShift Service on AWS:

```
$ oc get pods
```

You should see a line starting with **myapp-<number>-<hash>**, and that is your application running in Red Hat OpenShift Service on AWS.

5. Before your application is functional, you must initialize the database by running the database migration script. There are two ways you can do this:

- Manually from the running frontend container:
    - Exec into frontend container with **rsh** command:

    ```
    $ oc rsh <frontend_pod_id>
    ```

    - Run the migration from inside the container:

    ```
    $ RAILS_ENV=production bundle exec rake db:migrate
    ```

    If you are running your Rails application in a **development** or **test** environment you do not have to specify the **RAILS_ENV** environment variable.

- By adding pre-deployment lifecycle hooks in your template.

11.4.3. Creating a route for your application

You can expose a service to create a route for your application.