



OpenShift Container Platform 4.1

Images

Creating and managing images and imagestreams in OpenShift Container Platform
4.1

OpenShift Container Platform 4.1 Images

Creating and managing images and imagestreams in OpenShift Container Platform 4.1

Legal Notice

Copyright © 2019 Red Hat, Inc.

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

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

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

Abstract

This document provides instructions for creating and managing images and imagestreams in OpenShift Container Platform 4.1. It also provides instructions on using templates.

Table of Contents

CHAPTER 1. CONFIGURING THE SAMPLES OPERATOR	5
1.1. UNDERSTANDING THE SAMPLES OPERATOR	5
1.2. SAMPLES OPERATOR CONFIGURATION PARAMETERS	5
1.2.1. Configuration restrictions	7
1.2.2. Conditions	7
1.3. ACCESSING THE SAMPLES OPERATOR FILE	8
CHAPTER 2. UNDERSTANDING CONTAINERS, IMAGES, AND IMAGESTREAMS	9
2.1. IMAGES	9
2.2. CONTAINERS	9
2.3. IMAGE REGISTRY	10
2.4. IMAGE REPOSITORY	10
2.5. IMAGE TAGS	10
2.6. IMAGE IDS	10
2.7. USING IMAGESTREAMS	10
2.7.1. Imagestreamtags	11
2.8. IMAGESTREAM IMAGES	11
2.9. IMAGESTREAM TRIGGERS	12
2.10. ADDITIONAL RESOURCES	12
CHAPTER 3. CREATING IMAGES	13
3.1. LEARNING CONTAINER BEST PRACTICES	13
3.1.1. General container image guidelines	13
Reuse images	13
Maintain compatibility within tags	13
Avoid multiple processes	13
Use exec in wrapper scripts	13
Clean temporary files	14
Place instructions in the proper order	14
Mark important ports	15
Set environment variables	15
Avoid default passwords	15
Avoid sshd	15
Use volumes for persistent data	15
3.1.2. OpenShift Container Platform-specific guidelines	16
Enable images for source-to-image (S2I)	16
Support arbitrary user ids	16
Use services for inter-image communication	17
Provide common libraries	17
Use environment variables for configuration	17
Set image metadata	18
Clustering	18
Logging	18
Liveness and readiness probes	19
Templates	19
3.2. INCLUDING METADATA IN IMAGES	19
3.2.1. Defining image metadata	19
3.3. TESTING S2I IMAGES	20
3.3.1. Understanding testing requirements	21
3.3.2. Generating scripts and tools	21
3.3.3. Testing locally	21

3.3.4. Basic testing workflow	22
3.3.5. Using OpenShift Container Platform for building the image	22
CHAPTER 4. MANAGING IMAGESTREAMS	24
4.1. USING IMAGESTREAMS	24
4.2. CONFIGURING IMAGESTREAMS	25
4.3. IMAGESTREAM IMAGES	26
4.4. IMAGESTREAMTAGS	26
4.5. IMAGESTREAM CHANGE TRIGGERS	27
4.6. IMAGESTREAM MAPPING	27
4.7. WORKING WITH IMAGESTREAMS	30
4.7.1. Getting information about imagestreams	30
4.7.2. Adding tags to an imagestream	31
4.7.3. Adding tags for an external image	32
4.7.4. Updating imagestreamtags	32
4.7.5. Removing imagestreamtags	33
4.7.6. Configuring periodic importing of imagestreamtags	33
CHAPTER 5. IMAGE CONFIGURATION RESOURCES	34
5.1. IMAGE CONTROLLER CONFIGURATION PARAMETERS	34
5.2. CONFIGURING IMAGE SETTINGS	35
5.2.1. Importing insecure registries and blocking registries	36
CHAPTER 6. USING TEMPLATES	38
6.1. UNDERSTANDING TEMPLATES	38
6.2. UPLOADING A TEMPLATE	38
6.3. CREATING AN APPLICATION USING THE WEB CONSOLE	38
6.4. CREATING OBJECTS FROM TEMPLATES USING THE CLI	39
6.4.1. Adding labels	39
6.4.2. Listing parameters	39
6.4.3. Generating a list of objects	40
6.5. MODIFYING UPLOADED TEMPLATES	41
6.6. USING INSTANT APP AND QUICKSTART TEMPLATES	41
6.6.1. Quickstart templates	42
6.6.1.1. Web framework Quickstart templates	42
6.7. WRITING TEMPLATES	43
6.7.1. Writing the template description	43
6.7.2. Writing template labels	45
6.7.3. Writing template parameters	45
6.7.4. Writing the template object list	47
6.7.5. Marking a template as bindable	48
6.7.6. Exposing template object fields	48
6.7.7. Waiting for template readiness	50
6.7.8. Creating a template from existing objects	52
CHAPTER 7. USING RUBY ON RAILS	53
7.1. SETTING UP THE DATABASE	53
7.2. WRITING YOUR APPLICATION	54
7.2.1. Creating a welcome page	55
7.2.2. Configuring application for OpenShift Container Platform	55
7.2.3. Storing your application in Git	56
7.3. DEPLOYING YOUR APPLICATION TO OPENSIFT CONTAINER PLATFORM	57
7.3.1. Creating the database service	57
7.3.2. Creating the frontend service	58

7.3.3. Creating a route for your application	59
CHAPTER 8. USING IMAGES	60
8.1. USING IMAGES OVERVIEW	60
8.2. CONFIGURING JENKINS IMAGES	60
8.2.1. Configuration and customization	60
8.2.1.1. OpenShift Container Platform OAuth authentication	61
8.2.1.2. Jenkins authentication	61
8.2.2. Jenkins environment variables	62
8.2.3. Providing Jenkins cross project access	65
8.2.4. Jenkins cross volume mount points	66
8.2.5. Customizing the Jenkins image through Source-To-Image	66
8.2.6. Configuring the Jenkins Kubernetes plug-in	67
8.2.7. Jenkins permissions	69
8.2.8. Creating a Jenkins service from a template	70
8.2.9. Using the Jenkins Kubernetes plug-in	71
8.2.10. Jenkins memory requirements	73
8.2.11. Additional Resources	73
8.3. JENKINS AGENT	73
8.3.1. Jenkins agent images	74
8.3.2. Jenkins agent environment variables	74
8.3.3. Jenkins agent memory requirements	75
8.3.4. Jenkins agent Gradle builds	76
8.3.5. Jenkins agent pod retention	76

CHAPTER 1. CONFIGURING THE SAMPLES OPERATOR

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

Prerequisites

- Deploy an OpenShift Container Platform cluster.

1.1. UNDERSTANDING THE SAMPLES OPERATOR

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

The Samples Operator copies the pull secret that is captured by the installation program into the OpenShift namespace and names the secret, **samples-registry-credentials**, to facilitate imagestream imports from **registry.redhat.io**. Additionally, to facilitate imagestream 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 to facilitate image import.

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

The image for the Samples Operator contains imagestream and template definitions for the associated OpenShift Container Platform release. When each sample is created or updated, the Samples Operator includes an annotation that denotes the version of OpenShift Container Platform. The Operator uses this annotation to ensure that each sample matches the release version. Samples outside of its inventory are ignored, as are skipped samples. Modifications to any samples that are managed by the Operator, where that version annotation is modified or deleted, will be reverted automatically. The Jenkins images are actually part of the image payload from installation and are tagged into the imagestreams directly.


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

- Operator managed imagestreams.
- Operator managed templates.
- Operator generated configuration resources.
- Cluster status resources.
- The **samples-registry-credentials** secret.

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

1.2. SAMPLES OPERATOR CONFIGURATION PARAMETERS

The samples resource offers the following configuration fields:

Parameter	Description
Management State	<p>Managed: The Samples Operator updates the samples as the configuration dictates.</p> <p>Unmanaged: The Samples Operator ignores updates to its configuration resource object and any imagestreams or templates in the OpenShift namespace.</p> <p>Removed: The Samples Operator removes the set of Managed imagestreams and templates in the OpenShift namespace. It ignores new samples created by the cluster administrator or any samples in the skipped lists. After the removals are complete, the Samples Operator works like it is in the Unmanaged state and ignores any watch events on the sample resources, imagestreams, or templates.</p> <div>  <p>NOTE</p> <p>Neither deletion nor setting the Management State to Removed are completed while imagestream imports are still in progress. Once progress has completed, either in success or in error, the deletion or removal commences.</p> <p>Secret, imagestream, and template watch events are ignored once deletion or removal has started.</p> </div>
Samples Registry	<p>Overrides the registry from which images are imported.</p> <div>  <p>NOTE</p> <p>Creation or update of RHEL content does not commence if the secret for pull access is not in place when either Samples Registry is not explicitly set (i.e., the empty string), or when it is set to registry.redhat.io. In both cases, image imports will work off of registry.redhat.io, which requires credentials.</p> <p>Creation or update of RHEL content is not gated by the existence of the pull secret if the Samples Registry is overridden to a value other than the empty string or registry.redhat.io.</p> </div>
Architectures	Placeholder to choose an architecture type. Currently only x86 is supported.
Skipped Imagestreams	Imagestreams that are in the Samples Operator's inventory, but that the cluster administrator wants the Operator to ignore or not manage.
Skipped Templates	Templates that are in the Samples Operator's inventory, but that the cluster administrator wants the Operator to ignore or not manage.

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

1.2.1. Configuration restrictions

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

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

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

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

1.2.2. Conditions

The samples resource maintains the following conditions in its status:

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

1.3. ACCESSING THE SAMPLES OPERATOR FILE

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

Procedure

- Access the Samples Operator file:

```
# oc get configs.samples.operator.openshift.io -n openshift-cluster-samples-operator
```

The following is an example of the Samples Operator file:

```
apiVersion: samples.operator.openshift.io/v1
kind: SamplesResource
projectName: cluster-samples-operator
...
```

CHAPTER 2. UNDERSTANDING CONTAINERS, IMAGES, AND IMAGESTREAMS

Containers, images, and imagestreams 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 imagestream 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.

2.1. IMAGES

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

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

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

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

2.2. CONTAINERS

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

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

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

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

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

2.3. IMAGE REGISTRY

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

```
registry.redhat.io
```

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

2.4. IMAGE REPOSITORY

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

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

2.5. 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 imagestream. 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.

2.6. 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
```

2.7. USING IMAGESTREAMS

An imagestream and its associated tags provide an abstraction for referencing container images from within OpenShift Container Platform. The imagestream 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.

Imagestreams 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 imagestream 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 `imagestreamtag` 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 will continue using the previous, presumably known good image.

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

- OpenShift Container Platform's integrated registry.
- An external registry, for example **registry.redhat.io** or **hub.docker.com**.
- Other imagestreams in the OpenShift Container Platform cluster.

When you define an object that references an `imagestreamtag` (such as a Build or Deployment configuration), you point to an `imagestreamtag`, not the Docker repository. When you Build or Deploy your application, OpenShift Container Platform queries the Docker repository using the `imagestreamtag` to locate the associated ID of the image and uses that exact image.

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

Using imagestreams has several significant benefits:

- You can tag, rollback a tag, and quickly deal with images, without having to re-push using the command line.
- You can trigger Builds and Deployments when a new image is pushed to the registry. Also, OpenShift Container Platform has generic triggers for other resources, such as Kubernetes objects.
- You can mark a tag for periodic re-import. If the source image has changed, that change is picked up and reflected in the imagestream, which triggers the Build and/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 `imagestreamtag` will still point to a known-good version of the image, ensuring that your application will not break unexpectedly.
- You can configure security around who can view and use the images through permissions on the imagestream objects.
- Users that lack permission to read or list images on the cluster level can still retrieve the images tagged in a project using imagestreams.

2.7.1. Imagestreamtags

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

2.8. IMAGESTREAM IMAGES

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

2.9. IMAGESTREAM TRIGGERS

An imagestream trigger causes a specific action when an imagestreamtag 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.

2.10. ADDITIONAL RESOURCES

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

CHAPTER 3. 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 that can be used on OpenShift Container Platform. Once you have created an image, you can push it to the internal registry.

3.1. LEARNING CONTAINER BEST PRACTICES

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

3.1.1. General container image guidelines

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

Reuse images

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

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

Avoid multiple processes

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

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

Use **exec** in wrapper scripts

Many images use wrapper scripts to do some setup before starting a process for the software being run. If your image uses such a script, that script should use **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 will go to your wrapper

script instead of your software's process. This is not what you want, as illustrated here:

Say 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. Now say that you want to kill your container with **CTRL+C**. If your wrapper script used **exec** to start the server process, **podman** will send **SIGINT** to the server process, and everything will work as you expect. If you didn't use **exec** in your wrapper script, **podman** will send **SIGINT** to the process for the wrapper script and your process will keep 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, killing any child processes you may have launched from your **PID 1** process.

See the ["Docker and the PID 1 zombie reaping problem"](#) blog article for additional implications. Also see the ["Demystifying the init system \(PID 1\)"](#) blog article for a deep dive on **PID 1** and **init** systems.

Clean temporary files

All temporary files you create during the build process should be removed. This also includes any files added with the **ADD** command. For example, we strongly recommended that you 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 will 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 will show up under **podman ps** associated with containers created from your image
- Exposed ports will also be present in the metadata for your image returned by **podman inspect**
- Exposed ports will be 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

It is best to avoid setting default passwords. Many people will 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 should be configurable using an environment variable instead.

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

Avoid sshd

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

Use volumes for persistent data

Images should use a [volume](#) for persistent data. This way OpenShift Container Platform mounts the network storage to the node running the container, and if the container moves to a new node the storage is reattached to that node. By using the volume for all persistent storage needs, the content is

preserved even if the container is restarted or moved. If your image writes data to arbitrary locations within the container, that content might 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 will make it easier to take advantage of it later.

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

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



NOTE

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

Additional resources

- Docker documentation - [Best practices for writing Dockerfiles](#)
- Project Atomic documentation - [Guidance for Container Image Authors](#)

3.1.2. OpenShift Container Platform-specific guidelines

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

Enable images for source-to-image (S2I)

For images that are intended to run application code provided by a third party, such as a Ruby image designed to run Ruby code provided by a developer, you can enable your image to work with the [Source-to-Image \(S2I\)](#) build tool. S2I is a framework which 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.

For example, this [Python image](#) defines S2I scripts for building various versions of Python applications.

Support arbitrary user ids

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

For an image to support running as an arbitrary user, directories and files that may be written to by processes in the image should be owned by the root group and be read/writable by that group. Files to be executed should 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. The root group does not have any special permissions (unlike the root user) so there are no security concerns with this arrangement. 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.

Because the user ID of the container is generated dynamically, it will not have an associated entry in `/etc/passwd`. This can cause problems for applications that expect to be able to look up their user ID. One way to address this problem is to dynamically create a `passwd` file entry with the container's user ID as part of the image's start script. This is what a Dockerfile might include:

```
RUN chmod g=u /etc/passwd
ENTRYPOINT [ "uid_entrpoint" ]
USER 1001
```

Where `uid_entrpoint` contains:

```
if ! whoami &> /dev/null; then
  if [ -w /etc/passwd ]; then
    echo "${USER_NAME:-default}:x:${id -u}:0:${USER_NAME:-default} user:${HOME}:/sbin/nologin"
  >> /etc/passwd
fi
fi
```

For a complete example of this, see [this Dockerfile](#).

Lastly, the final **USER** declaration in the **Dockerfile** should specify the user ID (numeric value) and not the user name. This allows OpenShift Container Platform to validate the authority the image is attempting to run with and prevent running images that are trying to run as root, because running containers as a privileged user exposes potential security holes. If the image does not specify a **USER**, it inherits the **USER** from the parent image.



IMPORTANT

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

Use services for inter-image communication

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

Provide common libraries

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

Use environment variables for configuration

Users of your image should be able to configure it without having to create a downstream image based on your image. This means that the runtime configuration should be 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 should be defined in terms of environment variables that provide the service endpoint information. This allows an application to dynamically consume a datasource service that is defined in the OpenShift Container Platform environment without modifying the application image.

In addition, tuning should be 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 should 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.

See the following references for more on how to manage **cgroup** quotas in containers:

- Blog article - [Resource management in Docker](#)
- Docker documentation - [Runtime Metrics](#)
- Blog article - [Memory inside Linux containers](#)

Set image metadata

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

Clustering

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

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

Logging

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

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

Liveness and readiness probes

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

Templates

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

Additional resources

- [Docker basics](#)
- [Dockerfile reference](#)
- [Project Atomic Guidance for Container Image Authors](#)

3.2. INCLUDING METADATA IN IMAGES

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

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

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

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

Table 3.1. Supported Metadata

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

3.3. TESTING S2I IMAGES

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

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

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

3.3.1. Understanding testing requirements

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

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

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

3.3.2. 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.

3.3.3. 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
```

3.3.4. 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.

3.3.5. Using OpenShift Container Platform for building the image

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

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

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

CHAPTER 4. MANAGING IMAGESTREAMS

Imagestreams 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.

4.1. USING IMAGESTREAMS

An imagestream and its associated tags provide an abstraction for referencing container images from within OpenShift Container Platform. The imagestream 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.

Imagestreams 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 imagestream 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 imagestreamtag 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 will continue using the previous, presumably known good image.

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

- OpenShift Container Platform's integrated registry.
- An external registry, for example **registry.redhat.io** or **hub.docker.com**.
- Other imagestreams in the OpenShift Container Platform cluster.

When you define an object that references an imagestreamtag (such as a Build or Deployment configuration), you point to an imagestreamtag, not the Docker repository. When you Build or Deploy your application, OpenShift Container Platform queries the Docker repository using the imagestreamtag to locate the associated ID of the image and uses that exact image.

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

Using imagestreams has several significant benefits:

- You can tag, rollback a tag, and quickly deal with images, without having to re-push using the command line.
- You can trigger Builds and Deployments when a new image is pushed to the registry. Also, OpenShift Container Platform has generic triggers for other resources, such as Kubernetes objects.
- You can mark a tag for periodic re-import. If the source image has changed, that change is picked up and reflected in the imagestream, which triggers the Build and/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 imagestreamtag will still point to a known-good version of the image, ensuring that your application will not break unexpectedly.
- You can configure security around who can view and use the images through permissions on the imagestream objects.
- Users that lack permission to read or list images on the cluster level can still retrieve the images tagged in a project using imagestreams.

4.2. CONFIGURING IMAGESTREAMS

An imagestream object file contains the following elements.

Imagestream object definition

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

- ❶ The name of the imagestream.
- ❷ Docker repository path where new images can be pushed to add/update them in this imagestream.
- ❸ The SHA identifier that this imagestreamtag currently references. Resources that reference this imagestreamtag use this identifier.

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

4.3. IMAGESTREAM IMAGES

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

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

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

The imagestream image consists of the imagestream 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 imagestream image looks like:

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

4.4. IMAGESTREAMTAGS

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

Imagestreamtags 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 will be available at the second position, and so forth. This allows for easy rollbacks to make tags point to historical images again.

The following imagestreamtag is from an imagestream object:

Imagestreamtag with two images in its history

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

```

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

```

Imagestreamtags 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 imagestreamtag and could be updated in the future to change which image they follow, much like a symlink. Note that these new levels are not guaranteed to be backwards-compatible.
For example, the **latest** imagestreamtags that ship with OpenShift Container Platform are tracking tags. This means consumers of the **latest** imagestreamtag will be updated to the newest level of the framework provided by the image when a new level becomes available. A **latest** imagestreamtag to **v3.10** could 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 imagestreamtag, which might not be the latest version of an image. For example, if the **latest** imagestreamtag 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** imagestreamtag.



NOTE

Tracking tags are limited to a single imagestream and cannot reference other imagestreams.

You can create your own imagestreamtags for your own needs.

The imagestreamtag is composed of the name of the imagestream 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 imagestreamtag would be:

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

4.5. IMAGESTREAM CHANGE TRIGGERS

Imagestream 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.

4.6. IMAGESTREAM MAPPING

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

**NOTE**

Configuring imagestream mappings is an advanced feature.

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

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

Imagestream mapping object definition

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



```

- ENV=/opt/app-root/etc/scl_enable
- PROMPT_COMMAND=. /opt/app-root/etc/scl_enable
- RUBY_VERSION=2.2
ExposedPorts:
  8080/tcp: {}
Labels:
  build-date: 2015-12-23
  io.k8s.description: Platform for building and running Ruby 2.2 applications
  io.k8s.display-name: 172.30.56.218:5000/test/origin-ruby-sample:latest
  io.openshift.build.commit.author: Ben Parees <bparees@users.noreply.github.com>
  io.openshift.build.commit.date: Wed Jan 20 10:14:27 2016 -0500
  io.openshift.build.commit.id: 00cad392d39d5ef9117cbc8a31db0889eedd442
  io.openshift.build.commit.message: 'Merge pull request #51 from php-coder/fix_url_and_sti'
  io.openshift.build.commit.ref: master
  io.openshift.build.image: centos/ruby-22-
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b28e
  io.openshift.build.source-location: https://github.com/openshift/ruby-hello-world.git
  io.openshift.builder-base-version: 8d95148
  io.openshift.builder-version: 8847438ba06307f86ac877465eadc835201241df
  io.openshift.s2i.scripts-url: image:///usr/libexec/s2i
  io.openshift.tags: builder,ruby,ruby22
  io.s2i.scripts-url: image:///usr/libexec/s2i
  license: GPLv2
  name: CentOS Base Image
  vendor: CentOS
User: "1001"
WorkingDir: /opt/app-root/src
Container: 86e9a4a3c760271671ab913616c51c9f3cea846ca524bf07c04a6f6c9e103a76
ContainerConfig:
  AttachStdout: true
  Cmd:
    - /bin/sh
    - -c
    - tar -C /tmp -xf - && /usr/libexec/s2i/assemble
  Entrypoint:
    - container-entrypoint
  Env:
    - RACK_ENV=production
    - OPENSIFT_BUILD_NAME=ruby-sample-build-1
    - OPENSIFT_BUILD_NAMESPACE=test
    - OPENSIFT_BUILD_SOURCE=https://github.com/openshift/ruby-hello-world.git
    - EXAMPLE=sample-app
    - PATH=/opt/app-root/src/bin:/opt/app-
root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/sbin:/bin
    - STI_SCRIPTS_URL=image:///usr/libexec/s2i
    - STI_SCRIPTS_PATH=/usr/libexec/s2i
    - HOME=/opt/app-root/src
    - BASH_ENV=/opt/app-root/etc/scl_enable
    - ENV=/opt/app-root/etc/scl_enable
    - PROMPT_COMMAND=. /opt/app-root/etc/scl_enable
    - RUBY_VERSION=2.2
  ExposedPorts:
    8080/tcp: {}
  Hostname: ruby-sample-build-1-build
  Image: centos/ruby-22-
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b28e

```

```

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

```

4.7. WORKING WITH IMAGESTREAMS

The following sections describe how to use imagestreams and imagestreamtags.

4.7.1. Getting information about imagestreams

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

Procedure

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

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

For example:

```

$ oc describe is/python

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

3.5
  tagged from centos/python-35-centos7

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

```

- Get all the information available about particular imagestreamtag:

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

For example:

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

```
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-entrpoint
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.

4.7.2. Adding tags to an imagestream

You can add additional tags to imagestreams.

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
```

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

- Confirm the imagestream 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
```

```
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

```

4.7.3. Adding tags for an external image

You can add tags for external images.

Procedure

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

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

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

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

4.7.4. Updating imagestreamtags

You can update a tag to reflect another tag in an imagestream.

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 imagestream:

```
$ oc tag python:3.6 python:latest
Tag python:latest set to
python@sha256:438208801c4806548460b27bd1fbc7bb188273d13871ab43f.
```

4.7.5. Removing imagestreamtags

You can remove old tags from an imagestream.

Procedure

- Remove old tags from an imagestream:

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

For example:

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

Deleted tag default/python:3.5.

4.7.6. Configuring periodic importing of imagestreamtags

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
```

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

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

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

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

CHAPTER 5. IMAGE CONFIGURATION RESOURCES

Use the following procedure to configure image registries.

5.1. IMAGE CONTROLLER CONFIGURATION PARAMETERS

The **image.config.openshift.io/cluster** resource offers the following configuration parameters.

Parameter	Description
Image	<p>Holds cluster-wide information about how to handle images. The canonical, and only valid name is cluster.</p> <p>spec: Holds user-settable values for configuration. You can edit the spec subsection.</p> <p>status: Holds observed values from the cluster.</p>
ImageSpec	<p>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 will have the appropriate permissions.</p> <p>additionalTrustedCA: A reference to a ConfigMap containing additional CAs that should be trusted during ImageStream import, pod image pull, openshift-image-registry pullthrough, and builds.</p> <p>The namespace for this ConfigMap is openshift-config. The format of the ConfigMap is to use the registry hostname as the key, and the PEM-encoded certificate as the value, for each additional registry CA to trust.</p> <p>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.</p>
ImageStatus	<p>internalRegistryHostname: Set by the Image Registry Operator, which controls the internalRegistryHostname. It sets the hostname for the default internal image registry. The value must be in hostname[:port] format. For backward compatibility, you can still use the OPENSHIFT_DEFAULT_REGISTRY environment variable, but this setting overrides the environment variable.</p> <p>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 ImageStreams. The value must be in hostname[:port] format.</p>

Parameter	Description
RegistryLocation	<p>Contains a location of the registry specified by the registry domain name. The domain name might include wildcards.</p> <p>domainName: Specifies a domain name for the registry. In case the registry uses a non-standard (80 or 443) port, the port should be included in the domain name as well.</p> <p>insecure: Insecure indicates whether the registry is secure or insecure. By default, if not otherwise specified, the registry is assumed to be secure.</p>
RegistrySources	<p>Holds cluster-wide information about how to handle the registries config.</p> <p>insecureRegistries: Registries which do not have a valid TLS certificate or only support HTTP connections.</p> <p>blockedRegistries: Blacklisted for image pull and push actions. All other registries are allowed.</p> <p>allowedRegistries: Whitelisted for image pull and push actions. All other registries are blocked.</p> <p>Only one of blockedRegistries or allowedRegistries may be set</p>

5.2. CONFIGURING IMAGE SETTINGS

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

Procedure

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

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

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

```
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
  insecureRegistries: 5
    - insecure.com
  blockedRegistries: 6
    - untrusted.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 will have the appropriate permissions.
- 3 **additionalTrustedCA:** A reference to a ConfigMap containing additional CAs that should be trusted during **ImageStream import, pod image pull, openshift-image-registry pullthrough**, and builds. The namespace for this ConfigMap is **openshift-config**. The format of the ConfigMap is to use the registry hostname as the key, and the base64-encoded certificate as the value, for each additional registry CA to trust.
- 4 **registrySources:** Contains configuration that determines how the container runtime should treat individual registries when accessing images for builds and pods. For instance, whether or not to allow insecure access. It does not contain configuration for the internal cluster registry.
- 5 **insecureRegistries:** Registries which do not have a valid TLS certificate or only support HTTP connections.
- 6 **blockedRegistries:** Blacklisted for image pull and push actions. All other registries are allowed.

5.2.1. Importing insecure registries and blocking registries

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

Insecure external registries, such as those do not have a valid TLS certificate or only support HTTP connections, should be avoided.

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** resource:


```

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:
    insecureRegistries: 1
    - insecure.com
    blockedRegistries: 2
    - untrusted.com
status:
  internalRegistryHostname: image-registry.openshift-image-registry.svc:5000

```

- 1 Specify an insecure registry.
- 2 Specify any registries that should be blacklisted for image pull and push actions. All other registries are allowed.

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

```

cat /host/etc/containers/registries.conf
[registries]
[registries.search]
  registries = ["registry.access.redhat.com", "docker.io"]
[registries.insecure]
  registries = ["insecure.com"]
[registries.block]
  registries = ["untrusted.com"]

```

CHAPTER 6. USING TEMPLATES

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

6.1. UNDERSTANDING TEMPLATES

A template describes a set of objects that can be parameterized and processed to produce a list of objects for creation by OpenShift Container Platform. A template can be processed to create anything you have permission to create within a project, for example services, build configurations, and DeploymentConfigs. A template may 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.

6.2. UPLOADING A TEMPLATE

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

Procedure

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

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

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

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

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

6.3. CREATING AN APPLICATION USING THE WEB CONSOLE

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

Procedure

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



NOTE

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

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

```

metadata:
  name: "ruby"
  creationTimestamp: null
spec:
  dockerImageRepository: "registry.redhat.io/openshift3/ruby-20-rhel7"
  tags:
    -
      name: "2.0"
      annotations:
        description: "Build and run Ruby 2.0 applications"
        iconClass: "icon-ruby"
        tags: "builder,ruby" 1
        supports: "ruby:2.0,ruby"
        version: "2.0"

```

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

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

6.4. CREATING OBJECTS FROM TEMPLATES USING THE CLI

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

6.4.1. 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
```

6.4.2. Listing parameters

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

Procedure

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

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

Alternatively, if the template is already uploaded:

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

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

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

6.4.3. 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 may appear 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=-
```

6.5. 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>
```

6.6. USING INSTANT APP AND QUICKSTART TEMPLATES

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

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

Procedure

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

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

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



NOTE

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

6.6.1. Quickstart templates

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

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

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

6.6.1.1. Web framework Quickstart templates

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

- CakePHP: a PHP web framework (includes a MySQL database)

- **Dancer:** a Perl web framework (includes a MySQL database)
- **Django:** a Python web framework (includes a PostgreSQL database)
- **NodeJS:** a NodeJS web application (includes a MongoDB database)
- **Rails:** a Ruby web framework (includes a PostgreSQL database)

6.7. WRITING TEMPLATES

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

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

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

6.7.1. Writing the template description

The template description informs users what the template does and helps them find it when searching in the web console. Additional metadata beyond the template name is optional, but useful to have. In addition to general descriptive information, the metadata also includes a set of tags. Useful tags include the name of the language the template is related to (for example, **java**, **php**, **ruby**, and so on).

The following is an example of template description metadata:

```

kind: Template
apiVersion: v1
metadata:
  name: cakephp-mysql-example ❶
  annotations:
    openshift.io/display-name: "CakePHP MySQL Example (Ephemeral)" ❷
  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." ❸
  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. ❹
  tags: "quickstart,php,cakephp" ❺
  iconClass: icon-php ❻
  openshift.io/provider-display-name: "Red Hat, Inc." ❼
  openshift.io/documentation-url: "https://github.com/sclorg/cakephp-ex" ❽
  openshift.io/support-url: "https://access.redhat.com" ❾
message: "Your admin credentials are ${ADMIN_USERNAME}:${ADMIN_PASSWORD}" ❿

```

- ❶ The unique name of the template.
- ❷ A brief, user-friendly name, which can be employed by user interfaces.
- ❸ A description of the template. Include enough detail that the user will 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.
- ❹ Additional template description. This may be displayed by the service catalog, for example.
- ❺ Tags to be associated with the template for searching and grouping. Add tags that will include it into one of the provided catalog categories. Refer to the **id** and **categoryAliases** in **CATALOG_CATEGORIES** in the console's constants file. The categories can also be customized for the whole cluster.
- ❻ An icon to be displayed with your template in the web console. Choose from our existing logo icons when possible. You can also use icons from FontAwesome and PatternFly. Alternatively, provide icons through CSS customizations that can be added to an OpenShift Container Platform cluster that uses your template. You must specify an icon class that exists, or it will prevent falling back to the generic icon.
- ❼ The name of the person or organization providing the template.
- ❽ A URL referencing further documentation for the template.
- ❾ A URL where support can be obtained for the template.
- ❿ 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.

6.7.2. Writing template labels

Templates can include a set of labels. These labels will be 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" ❶
  app: "${NAME}" ❷
```

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

6.7.3. Writing template parameters

Parameters allow a value to be supplied by the user 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 the user to supply a host name or other user-specific value that is required to customize the template. Parameters can be referenced in two ways:

- As a string value by placing values in the form **`${PARAMETER_NAME}`** in any string field in the template.
- As a json/yaml value by placing values in the form **`${{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 will be substituted and the resulting value will be a quoted string.

When using the **`${{PARAMETER_NAME}}`** syntax only a single parameter reference is allowed and leading/trailing characters are not permitted. The resulting value will be 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 will be 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 the user does 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 will generate 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`, 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}`.

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

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

```

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 will be replaced with the value of the **SOURCE_REPOSITORY_URL** parameter when the template is instantiated.
- 2 This value will be 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 will be 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. Don't make this a duplicate of the display name.
- 6 A default value for the parameter which will be used if the user does 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 the user cannot override it with an empty value. If the parameter does not provide a default or generated value, the user must supply a value.
- 8 A parameter which has its value generated.
- 9 The input to the generator. In this case, the generator will produce a 40 character alphanumeric value including upper and lowercase characters.
- 10 Parameters can be included in the template message. This informs the user about generated values.

6.7.4. Writing the template object list

The main portion of the template is the list of objects which will be created when the template is instantiated. This can be any valid API object, such as a **BuildConfig**, **DeploymentConfig**, **Service**, etc. The object will be created exactly as defined here, with any parameter values substituted in prior to creation. The definition of these objects can reference parameters defined earlier.

The following is an example of an object list:

```

kind: "Template"
apiVersion: "v1"
metadata:

```

```

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

```

1 The definition of a **Service** which will be created by this template.



NOTE

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

6.7.5. Marking a template as bindable

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

Procedure

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

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

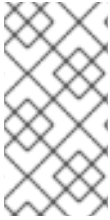
6.7.6. Exposing template object fields

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

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

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

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

**NOTE**

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

**NOTE**

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

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

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

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

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

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

Procedure

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

6.7.7. Waiting for template readiness

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

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

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

Template instantiation will 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 will fail.

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 ReplicaSet and deployment available (this honors readiness probes defined on the object)	Object reports Progressing condition as false

Kind	Readiness	Failure
DeploymentConfig	Object reports new ReplicationController and deployment available (this honors readiness probes defined on the object)	Object reports Progressing condition as false
Job	Object reports completion	Object reports that one or more failures have occurred
StatefulSet	Object reports all replicas ready (this honors readiness probes defined on the object)	Not applicable

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

```

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

```

Additional recommendations

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

- Avoid referencing the **latest** tag from images if that tag is used across major versions. This may 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.

6.7.8. Creating a template from existing objects

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

Procedure

1. Export objects in a project in YAML form:

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

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

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

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

CHAPTER 7. USING RUBY ON RAILS

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



WARNING

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

Prerequisites

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

7.1. SETTING UP THE DATABASE

Rails applications are almost always used with a database. For the 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 will create the **/var/lib/pgsql/data** directory, in which the data will be 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.

7.2. 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
Successfully installed rails-4.2.0
1 gem installed
```

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

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

3. Change into your new application directory:

```
$ cd rails-app
```

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

```
gem 'pg'
```

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

```
$ bundle install
```

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

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

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

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

```
$ rake db:create
```

This will create **development** and **test** database in your PostgreSQL server.

7.2.1. Creating a welcome page

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

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

- Create a **controller** with an index action
- Create a **view** page for the **welcome** controller **index** action
- Create a **route** that will serve applications root page with the created **controller** and **view**

Rails offers a generator that will do 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 don't see the page, check the logs that are output to your server to debug.

7.2.2. Configuring application for OpenShift Container Platform

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

Procedure

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

```
<% 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"] %>
```

7.2.3. Storing your application in Git

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

Prerequisites

- Install git.

Procedure

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

```
$ ls -l
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.

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

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

2. Push your application to your remote git repository.

```
$ git push
```

7.3. DEPLOYING YOUR APPLICATION TO OPENSIFT CONTAINER PLATFORM

You can deploy your application to OpenShift Container Platform.

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

Deploying your application in OpenShift Container Platform involves three steps:

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

Procedure

- To deploy your Ruby on Rails application, create a new Project for the application:

```
$ oc new-project rails-app --description="My Rails application" --display-name="Rails Application"
```

7.3.1. Creating the database service

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

To create the database service you will use the **oc new-app** command. To this command you must pass some necessary environment variables which will be 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
```

7.3.2. Creating the frontend service

To bring your application to OpenShift Container Platform, you must specify a repository in which your application lives.

Procedure

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

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

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

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

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

You should see the following section:

```
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. Once the build is complete, look at the running pods in OpenShift Container Platform:

```
$ oc get pods
```

You should see a line starting with **myapp-`<number>`--`<hash>`**, and that is your application running in OpenShift Container Platform.

5. Before your application will be 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 don't have to specify the **RAILS_ENV** environment variable.

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

7.3.3. Creating a route for your application

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

Procedure

- To expose a service by giving it an externally-reachable hostname like **www.example.com** use OpenShift Container Platform route. In your case you need to expose the frontend service by typing:

```
$ oc expose service rails-app --hostname=www.example.com
```



WARNING

Ensure the hostname you specify resolves into the IP address of the router.

CHAPTER 8. USING IMAGES

8.1. USING IMAGES OVERVIEW

Use the following topics to discover the different Source-to-Image (S2I), database, and other container images that are available for OpenShift Container Platform users.

Red Hat's official container images are provided in the Red Hat Registry at registry.redhat.io. OpenShift Container Platform's supported S2I, database, and Jenkins images are provided in the **openshift4** repository in the Red Hat Quay Registry. For example, **quay.io/openshift-release-dev/ocp-v4.0-
<address>** is the name of the OpenShift Application Platform image.

The xPaaS middleware images are provided in their respective product repositories on the Red Hat Registry but suffixed with a **-openshift**. For example, **registry.redhat.io/jboss-eap-6/eap64-openshift** is the name of the JBoss EAP image.

All Red Hat supported images covered in this section are described in the Red Hat Container Catalog. For every version of each image, you can find details on its contents and usage. Browse or search for the image that interests you.



IMPORTANT

The newer versions of container images are not compatible with earlier versions of OpenShift Container Platform. Verify and use the correct version of container images, based on your version of OpenShift Container Platform.

8.2. CONFIGURING JENKINS IMAGES

OpenShift Container Platform provides a container image for running Jenkins. This image provides a Jenkins server instance, which can be used to set up a basic flow for continuous testing, integration, and delivery.

This image also includes a sample Jenkins job, which triggers a new build of a BuildConfig that is defined in OpenShift Container Platform, tests the output of that build, and then on successful build, retags the output to indicate that the build is ready for production.

OpenShift Container Platform follows the [LTS](#) release of Jenkins. OpenShift Container Platform provides an image that contains Jenkins 2.x.

The OpenShift Container Platform Jenkins images are available on **quay.io** or **registry.redhat.io**.

Jenkins images are available through the Red Hat Registry:

```
$ docker pull registry.redhat.io/openshift4/ose-jenkins:<v4.1.4>
```

To use these images, you can either access them directly from these registries or push them into your OpenShift Container Platform container image registry. Additionally, you can create an ImageStream that points to the image, either in your container image registry or at the external location. Your OpenShift Container Platform resources can then reference the ImageStream.

8.2.1. Configuration and customization

You can manage Jenkins authentication in two ways:

- OpenShift Container Platform OAuth authentication provided by the OpenShift Login plug-in.
- Standard authentication provided by Jenkins.

8.2.1.1. OpenShift Container Platform OAuth authentication

OAuth authentication is activated by configuring options on the **Configure Global Security** panel in the Jenkins UI, or by setting the **OPENSIFT_ENABLE_OAUTH** environment variable on the Jenkins **Deployment configuration** to anything other than **false**. This activates the OpenShift Container Platform Login plug-in, which retrieves the configuration information from Pod data or by interacting with the OpenShift Container Platform API server.

Valid credentials are controlled by the OpenShift Container Platform identity provider.

Jenkins supports both browser and non-browser access.

Valid users are automatically added to the Jenkins authorization matrix at log in, where OpenShift Container Platform **Roles** dictate the specific Jenkins permissions the users have.

Users with the **admin** role have the traditional Jenkins administrative user permissions. Users with the **edit** or **view** role have progressively fewer permissions.



NOTE

The **admin** user that is pre-populated in the OpenShift Container Platform Jenkins image with administrative privileges is not given those privileges when OpenShift Container Platform OAuth is used. To grant these permissions the OpenShift Container Platform cluster administrator must explicitly define that user in the OpenShift Container Platform identity provider and assigns the **admin** role to the user.

Jenkins users' permissions that are stored can be changed after the users are initially established. The OpenShift Login plug-in polls the OpenShift Container Platform API server for permissions and updates the permissions stored in Jenkins for each user with the permissions retrieved from OpenShift Container Platform. If the Jenkins UI is used to update permissions for a Jenkins user, the permission changes are overwritten the next time the plug-in polls OpenShift Container Platform.

You can control how often the polling occurs with the **OPENSIFT_PERMISSIONS_POLL_INTERVAL** environment variable. The default polling interval is five minutes.

The easiest way to create a new Jenkins service using OAuth authentication is to use a template.

8.2.1.2. Jenkins authentication

Jenkins authentication is used by default if the image is run directly, without using a template.

The first time Jenkins starts, the configuration is created along with the administrator user and password. The default user credentials are **admin** and **password**. Configure the default password by setting the **JENKINS_PASSWORD** environment variable when using, and only when using, standard Jenkins authentication.

Procedure

- Create a Jenkins application that uses standard Jenkins authentication:

```
$ oc new-app -e \
  JENKINS_PASSWORD=<password> \
  openshift4/ose-jenkins
```

8.2.2. Jenkins environment variables

The Jenkins server can be configured with the following environment variables:

Variable	Definition	Example values and settings
OPENSIFT_ENABLE_OAUTH	Determines whether the OpenShift Login plug-in manages authentication when logging into Jenkins. To enable, set to true .	Default: false
JENKINS_PASSWORD	The password for the admin user when using standard Jenkins authentication. Not applicable when OPENSIFT_ENABLE_OAUTH is set to true .	Default: password
JAVA_MAX_HEAP_PARAMETER, CONTAINER_HEAP_PERCENT, JENKINS_MAX_HEAP_UPPER_BOUND_MB	<p>These values control the maximum heap size of the Jenkins JVM. If JAVA_MAX_HEAP_PARAMETER is set, its value takes precedence. Otherwise, the maximum heap size is dynamically calculated as CONTAINER_HEAP_PERCENT of the container memory limit, optionally capped at JENKINS_MAX_HEAP_UPPER_BOUND_MB MiB.</p> <p>By default, the maximum heap size of the Jenkins JVM is set to 50% of the container memory limit with no cap.</p>	<p>JAVA_MAX_HEAP_PARAMETER example setting: -Xmx512m</p> <p>CONTAINER_HEAP_PERCENT default: 0.5, or 50%</p> <p>JENKINS_MAX_HEAP_UPPER_BOUND_MB example setting: 512 MiB</p>
JAVA_INITIAL_HEAP_PARAMETER, CONTAINER_INITIAL_PERCENT	<p>These values control the initial heap size of the Jenkins JVM. If JAVA_INITIAL_HEAP_PARAMETER is set, its value takes precedence. Otherwise, the initial heap size is dynamically calculated as CONTAINER_INITIAL_PERCENT of the dynamically calculated maximum heap size.</p> <p>By default, the JVM sets the initial heap size.</p>	<p>JAVA_INITIAL_HEAP_PARAMETER example setting: -Xms32m</p> <p>CONTAINER_INITIAL_PERCENT example setting: 0.1, or 10%</p>

Variable	Definition	Example values and settings
CONTAINER_CORE_LIMIT	If set, specifies an integer number of cores used for sizing numbers of internal JVM threads.	Example setting: 2
JAVA_TOOL_OPTIONS	Specifies options to apply to all JVMs running in this container. It is not recommended to override this value.	Default: - XX:+UnlockExperimentalVMOptions - XX:+UseCGroupMemoryLimitForHeap - Dsun.zip.disableMemoryMapping=true
JAVA_GC_OPTS	Specifies Jenkins JVM garbage collection parameters. It is not recommended to override this value.	Default: - XX:+UseParallelGC - XX:MinHeapFreeRatio=5 - XX:MaxHeapFreeRatio=10 - XX:GCTimeRatio=4 - XX:AdaptiveSizePolicyWeight=90
JENKINS_JAVA_OVERRIDES	Specifies additional options for the Jenkins JVM. These options are appended to all other options, including the Java options above, and may be used to override any of them if necessary. Separate each additional option with a space; if any option contains space characters, escape them with a backslash.	Example settings: - Dfoo - Dbar ; - Dfoo=first\ value - Dbar=second\ value .
JENKINS_OPTS	Specifies arguments to Jenkins.	
INSTALL_PLUGINS	Specifies additional Jenkins plugins to install when the container is first run or when OVERRIDE_PV_PLUGINS_WITH_IMAGE_PLUGINS is set to true . Plug-ins are specified as a comma-delimited list of name:version pairs.	Example setting: git:3.7.0,subversion:2.10.2 .
OPENSIFT_PERMISSIONS_POLL_INTERVAL	Specifies the interval in milliseconds that the OpenShift Login plug-in polls OpenShift Container Platform for the permissions that are associated with each user that is defined in Jenkins.	Default: 300000 - 5 minutes

Variable	Definition	Example values and settings
OVERRIDE_PV_CONFIG_WITH_IMAGE_CONFIG	When running this image with an OpenShift Container Platform persistent volume for the Jenkins configuration directory, the transfer of configuration from the image to the Persistent Volume is performed only the first time the image starts because the Persistent Volume is assigned when the Persistent Volume Claim is created. If you create a custom image that extends this image and updates configuration in the custom image after the initial startup, the configuration is not copied over unless you set this environment variable to true .	Default: false
OVERRIDE_PV_PLUGINS_WITH_IMAGE_PLUGINS	When running this image with an OpenShift Container Platform persistent volume for the Jenkins configuration directory, the transfer of plugins from the image to the Persistent Volume is performed only the first time the image starts because the Persistent Volume is assigned when the Persistent Volume Claim is created. If you create a custom image that extends this image and updates plug-ins in the custom image after the initial startup, the plug-ins are not copied over unless you set this environment variable to true .	Default: false
ENABLE_FATAL_ERROR_LOG_FILE	When running this image with an OpenShift Container Platform Persistent Volume Claim for the Jenkins configuration directory, this environment variable allows the fatal error log file to persist when a fatal error occurs. The fatal error file is saved at /var/lib/jenkins/logs .	Default: false

Variable	Definition	Example values and settings
NODEJS_SLAVE_IMAGE	Setting this value overrides the image that is used for the default NodeJS agent Pod configuration. A related imagestreamtag named jenkins-agent-nodejs is in the project. This variable must be set before Jenkins starts the first time for it to have an effect.	Default NodeJS agent image in Jenkins server: image-registry.openshift-image-registry.svc:5000/openshift/jenkins-agent-nodejs:latest
MAVEN_SLAVE_IMAGE	Setting this value overrides the image used for the default maven agent Pod configuration. A related imagestreamtag named jenkins-agent-maven is in the project. This variable must be set before Jenkins starts the first time for it to have an effect.	Default Maven agent image in Jenkins server: image-registry.openshift-image-registry.svc:5000/openshift/jenkins-agent-maven:latest

8.2.3. Providing Jenkins cross project access

If you are going to run Jenkins somewhere other than your same project, you must provide an access token to Jenkins to access your project.

Procedure

1. Identify the secret for the service account that has appropriate permissions to access the project Jenkins must access:

```
$ oc describe serviceaccount jenkins
Name:      default
Labels:    <none>
Secrets:   { jenkins-token-uyswp  }
           { jenkins-dockercfg-xcr3d }
Tokens:    jenkins-token-izv1u
           jenkins-token-uyswp
```

In this case the secret is named **jenkins-token-uyswp**.

2. Retrieve the token from the secret:

```
$ oc describe secret <secret name from above>
Name:      jenkins-token-uyswp
Labels:    <none>
Annotations:  kubernetes.io/service-account.name=jenkins,kubernetes.io/service-account.uid=32f5b661-2a8f-11e5-9528-3c970e3bf0b7
Type:       kubernetes.io/service-account-token
Data
====
ca.crt: 1066 bytes
token: eyJhbGc...<content cut>...wRA
```

The token parameter contains the token value Jenkins requires to access the project.

8.2.4. Jenkins cross volume mount points

The Jenkins image can be run with mounted volumes to enable persistent storage for the configuration:

- **/var/lib/jenkins** - This is the data directory where Jenkins stores configuration files, including job definitions.

8.2.5. Customizing the Jenkins image through Source-To-Image

To customize the official OpenShift Container Platform Jenkins image, you can use the image as a Source-To-Image (S2I) builder.

You can use S2I to copy your custom Jenkins Jobs definitions, add additional plug-ins, or replace the provided **config.xml** file with your own, custom, configuration.

To include your modifications in the Jenkins image, you must have a Git repository with the following directory structure:

plugins

This directory contains those binary Jenkins plug-ins you want to copy into Jenkins.

plugins.txt

This file lists the plug-ins you want to install using the the following syntax:

```
pluginId:pluginVersion
```

configuration/jobs

This directory contains the Jenkins job definitions.

configuration/config.xml

This file contains your custom Jenkins configuration.

The contents of the **configuration/** directory is copied to the **/var/lib/jenkins/** directory, so you can also include additional files, such as **credentials.xml**, there.

The following example build configuration customizes the Jenkins image in OpenShift Container Platform:

```
apiVersion: v1
kind: BuildConfig
metadata:
  name: custom-jenkins-build
spec:
  source: ❶
  git:
    uri: https://github.com/custom/repository
    type: Git
  strategy: ❷
    sourceStrategy:
      from:
        kind: ImageStreamTag
        name: jenkins:2
        namespace: openshift
```

```

type: Source
output:
  to:
    kind: ImageStreamTag
    name: custom-jenkins:latest

```

- 1 The **source** parameter defines the source Git repository with the layout described above.
- 2 The **strategy** parameter defines the original Jenkins image to use as a source image for the build.
- 3 The **output** parameter defines the resulting, customized Jenkins image that you can use in deployment configurations instead of the official Jenkins image.

8.2.6. Configuring the Jenkins Kubernetes plug-in

The OpenShift Container Platform Jenkins image includes the pre-installed [Kubernetes plug-in](#) that allows Jenkins agents to be dynamically provisioned on multiple container hosts using Kubernetes and OpenShift Container Platform.

To use the Kubernetes plug-in, OpenShift Container Platform provides images that are suitable for use as Jenkins agents: the **Base**, **Maven**, and **Node.js** images.

Both the Maven and Node.js agent images are automatically configured as Kubernetes Pod Template images within the OpenShift Container Platform Jenkins image's configuration for the Kubernetes plug-in. That configuration includes labels for each of the images that can be applied to any of your Jenkins jobs under their **Restrict where this project can be run** setting. If the label is applied, jobs run under an OpenShift Container Platform Pod running the respective agent image.

The Jenkins image also provides auto-discovery and auto-configuration of additional agent images for the Kubernetes plug-in.

With the OpenShift Container Platform Sync plug-in, the Jenkins image on Jenkins start-up searches for the following within the project that it is running or the projects specifically listed in the plug-in's configuration:

- Imagestreams that have the label **role** set to **jenkins-slave**.
- Imagestreamtags that have the annotation **role** set to **jenkins-slave**.
- ConfigMaps that have the label **role** set to **jenkins-slave**.

When it finds an imagestream with the appropriate label, or imagestreamtag with the appropriate annotation, it generates the corresponding Kubernetes plug-in configuration so you can assign your Jenkins jobs to run in a Pod that runs the container image that is provided by the imagestream.

The name and image references of the imagestream or imagestreamtag are mapped to the name and image fields in the Kubernetes plug-in Pod template. You can control the label field of the Kubernetes plug-in Pod template by setting an annotation on the imagestream or imagestreamtag object with the key **slave-label**. Otherwise, the name is used as the label.

**NOTE**

Do not log into the Jenkins console and modify the Pod Template configuration. If you do so after the Pod Template is created, and the OpenShift Sync plug-in detects that the image associated with the ImageStream or ImageStreamTag has changed, it replaces the Pod Template and overwrites those configuration changes. You cannot merge a new configuration with the existing configuration.

Consider the ConfigMap approach if you have more complex configuration needs.

When it finds a ConfigMap with the appropriate label, it assumes that any values in the key-value data payload of the ConfigMap contains XML that is consistent with the configuration format for Jenkins and the Kubernetes plug-in Pod templates. A key differentiator to note when using ConfigMaps, instead of imagestreams or imagestreamtags, is that you can control all the parameters of the Kubernetes plug-in Pod template.

Example ConfigMap for **jenkins-agent**:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: jenkins-agent
  labels:
    role: jenkins-slave
data:
  template1: |-
    <org.csanchez.jenkins.plugins.kubernetes.PodTemplate>
      <inheritFrom></inheritFrom>
      <name>template1</name>
      <instanceCap>2147483647</instanceCap>
      <idleMinutes>0</idleMinutes>
      <label>template1</label>
      <serviceAccount>jenkins</serviceAccount>
      <nodeSelector></nodeSelector>
      <volumes/>
      <containers>
        <org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
          <name>jnlp</name>
          <image>openshift/jenkins-agent-maven-35-centos7:v3.10</image>
          <privileged>false</privileged>
          <alwaysPullImage>true</alwaysPullImage>
          <workingDir>/tmp</workingDir>
          <command></command>
          <args>${computer.jnlpMac} ${computer.name}</args>
          <ttyEnabled>false</ttyEnabled>
          <resourceRequestCpu></resourceRequestCpu>
          <resourceRequestMemory></resourceRequestMemory>
          <resourceLimitCpu></resourceLimitCpu>
          <resourceLimitMemory></resourceLimitMemory>
          <envVars/>
        </org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
      </containers>
    <envVars/>
    <annotations/>
```



```
<imagePullSecrets/>
<nodeProperties/>
</org.csanchez.jenkins.plugins.kubernetes.PodTemplate>
```



NOTE

If you log into the Jenkins console and make further changes to the Pod Template configuration after the Pod Template is created, and the OpenShift Sync plug-in detects that the ConfigMap has changed, it will replace the Pod Template and overwrite those configuration changes. You cannot merge a new configuration with the existing configuration.

Do not log into the Jenkins console and modify the Pod Template configuration. If you do so after the Pod Template is created, and the OpenShift Sync plug-in detects that the image associated with the ImageStream or ImageStreamTag has changed, it replaces the Pod Template and overwrites those configuration changes. You cannot merge a new configuration with the existing configuration.

Consider the ConfigMap approach if you have more complex configuration needs.

After it is installed, the OpenShift Sync plug-in monitors the API server of OpenShift Container Platform for updates to **ImageStreams**, **ImageStreamTags**, and **ConfigMaps** and adjusts the configuration of the Kubernetes plug-in.

The following rules apply:

- Removing the label or annotation from the **ConfigMap**, **ImageStream**, or **ImageStreamTag** results in the deletion of any existing **PodTemplate** from the configuration of the Kubernetes plug-in.
- If those objects are removed, the corresponding configuration is removed from the Kubernetes plug-in.
- Either creating appropriately labeled or annotated **ConfigMap**, **ImageStream**, or **ImageStreamTag** objects, or the adding of labels after their initial creation, leads to creating of a **PodTemplate** in the Kubernetes-plugin configuration.
- In the case of the **PodTemplate** by **ConfigMap** form, changes to the **ConfigMap** data for the **PodTemplate** are applied to the **PodTemplate** settings in the Kubernetes plug-in configuration and overrides any changes that were made to the **PodTemplate** through the Jenkins UI between changes to the **ConfigMap**.

To use a container image as a Jenkins agent, the image must run the slave agent as an entrypoint. For more details about this, refer to the official [Jenkins documentation](#).

8.2.7. Jenkins permissions

If in the ConfigMap the **<serviceAccount>** element of the Pod Template XML is the OpenShift Container Platform Service Account used for the resulting Pod, the service account credentials are mounted into the Pod. The permissions are associated with the service account and control which operations against the OpenShift Container Platform master are allowed from the Pod.

Consider the following scenario with service accounts used for the Pod, which is launched by the Kubernetes Plug-in that runs in the OpenShift Container Platform Jenkins image:

If you use the example template for Jenkins that is provided by OpenShift Container Platform, the **jenkins** service account is defined with the **edit** role for the project Jenkins runs in, and the master Jenkins Pod has that service account mounted.

The two default Maven and NodeJS Pod Templates that are injected into the Jenkins configuration are also set to use the same service account as the Jenkins master.

- Any Pod templates that are automatically discovered by the OpenShift Sync plug-in because their imagestreams or imagestreamtags have the required label or annotations are configured to use the Jenkins master's service account as their service account.
- For the other ways you can provide a Pod Template definition into Jenkins and the Kubernetes plug-in, you have to explicitly specify the service account to use. Those other ways include the Jenkins console, the **podTemplate** pipeline DSL that is provided by the Kubernetes plug-in, or labeling a ConfigMap whose data is the XML configuration for a Pod Template.
- If you do not specify a value for the service account, the **default** service account is used.
- Ensure that whatever service account is used has the necessary permissions, roles, and so on defined within OpenShift Container Platform to manipulate whatever projects you choose to manipulate from the within the Pod.

8.2.8. Creating a Jenkins service from a template

Templates provide parameter fields to define all the environment variables with predefined default values. OpenShift Container Platform provides templates to make creating a new Jenkins service easy. The Jenkins templates should be registered in the default **openshift** project by your cluster administrator during the initial cluster setup.

The two available templates both define deployment configuration and a service. The templates differ in their storage strategy, which affects whether or not the Jenkins content persists across a Pod restart.



NOTE

A Pod might be restarted when it is moved to another node or when an update of the deployment configuration triggers a redeployment.

- **jenkins-ephemeral** uses ephemeral storage. On Pod restart, all data is lost. This template is only useful for development or testing.
- **jenkins-persistent** uses a Persistent Volume store. Data survives a Pod restart.

To use a Persistent Volume store, the cluster administrator must define a Persistent Volume pool in the OpenShift Container Platform deployment.

After you select which template you want, you must instantiate the template to be able to use Jenkins.

Procedure

1. Create a new Jenkins application using one of the following methods:

- A Persistent Volume:

```
$ oc new-app jenkins-persistent
```

- Or an **emptyDir** type volume where configuration does not persist across Pod restarts:

```
$ oc new-app jenkins-ephemeral
```

8.2.9. Using the Jenkins Kubernetes plug-in

In the following example, the **openshift-jee-sample** BuildConfig causes a Jenkins Maven agent Pod to be dynamically provisioned. The Pod clones some Java source code, builds a WAR file, and causes a second BuildConfig, **openshift-jee-sample-docker** to run. The second BuildConfig layers the new WAR file into a container image.

The following example is a BuildConfig that uses the Jenkins Kubernetes plug-in.

```
kind: List
apiVersion: v1
items:
- kind: ImageStream
  apiVersion: v1
  metadata:
    name: openshift-jee-sample
- kind: BuildConfig
  apiVersion: v1
  metadata:
    name: openshift-jee-sample-docker
  spec:
    strategy:
      type: Docker
    source:
      type: Docker
      dockerfile: |-
        FROM openshift/wildfly-101-centos7:latest
        COPY ROOT.war /wildfly/standalone/deployments/ROOT.war
        CMD $STI_SCRIPTS_PATH/run
      binary:
        asFile: ROOT.war
    output:
      to:
        kind: ImageStreamTag
        name: openshift-jee-sample:latest
- kind: BuildConfig
  apiVersion: v1
  metadata:
    name: openshift-jee-sample
  spec:
    strategy:
      type: JenkinsPipeline
      jenkinsPipelineStrategy:
        jenkinsfile: |-
          node("maven") {
            sh "git clone https://github.com/openshift/openshift-jee-sample.git ."
            sh "mvn -B -Popenshift package"
            sh "oc start-build -F openshift-jee-sample-docker --from-file=target/ROOT.war"
          }
    triggers:
      - type: ConfigChange
```

It is also possible to override the specification of the dynamically created Jenkins agent Pod. The following is a modification to the previous example, which overrides the container memory and specifies an environment variable:

The following example is a BuildConfig that the Jenkins Kubernetes Plug-in, specifying memory limit and environment variable.

```
kind: BuildConfig
apiVersion: v1
metadata:
  name: openshift-jee-sample
spec:
  strategy:
    type: JenkinsPipeline
    jenkinsPipelineStrategy:
      jenkinsfile: |-
        podTemplate(label: "mypod", 1
          cloud: "openshift", 2
          inheritFrom: "maven", 3
          containers: [
            containerTemplate(name: "jnlp", 4
              image: "openshift/jenkins-agent-maven-35-centos7:v3.10", 5
              resourceRequestMemory: "512Mi", 6
              resourceLimitMemory: "512Mi", 7
              envVars: [
                envVar(key: "CONTAINER_HEAP_PERCENT", value: "0.25") 8
              ]
            ) {
          node("mypod") { 9
            sh "git clone https://github.com/openshift/openshift-jee-sample.git ."
            sh "mvn -B -Popenshift package"
            sh "oc start-build -F openshift-jee-sample-docker --from-file=target/ROOT.war"
          }
        }
      triggers:
        - type: ConfigChange
```

- 1 A new Pod template called **mypod** is defined dynamically. The new Pod template name is referenced in the node stanza.
- 2 The **cloud** value must be set to **openshift**.
- 3 The new Pod template can inherit its configuration from an existing Pod template. In this case, inherited from the Maven Pod template that is pre-defined by OpenShift Container Platform.
- 4 This example overrides values in the pre-existing Container, and must be specified by name. All Jenkins agent images shipped with OpenShift Container Platform use the Container name **jnlp**.
- 5 Specify the Container image name again. This is a known issue.
- 6 A memory request of **512 Mi** is specified.
- 7 A memory limit of **512 Mi** is specified.
- 8 An environment variable **CONTAINER_HEAP_PERCENT**, with value **0.25**, is specified.

- 9 The node stanza references the name of the defined Pod template.

By default, the pod is deleted when the build completes. This behavior can be modified with the plug-in or within a pipeline Jenkinsfile.

8.2.10. Jenkins memory requirements

When deployed by the provided Jenkins Ephemeral or Jenkins Persistent templates, the default memory limit is **1 Gi**.

By default, all other process that run in the Jenkins container cannot use more than a total of **512 MiB** of memory. If they require more memory, the container halts. It is therefore highly recommended that pipelines run external commands in a agent container wherever possible.

It is recommended to specify memory request and limit values on agent containers created by the Jenkins Kubernetes Plug-in. Admin users can set default values on a per-agent image basis through the Jenkins configuration. The memory request and limit parameters can also be overridden on a per-container basis.

You can increase the amount of memory available to Jenkins by overriding the **MEMORY_LIMIT** parameter when instantiating the Jenkins Ephemeral or Jenkins Persistent template.

8.2.11. Additional Resources

- See [Base image options](#) for more information on the [Red Hat Universal Base Images](#) (UBI).

8.3. JENKINS AGENT

OpenShift Container Platform provides three images that suitable for use as Jenkins agents: the **Base**, **Maven**, and **Node.js** images.

The first is a base image for Jenkins agents:

- It pulls in both the required tools, headless Java, the Jenkins JNLP client, and the useful ones including **git**, **tar**, **zip**, and **nss** among others.
- It establishes the JNLP agent as the entrypoint.
- It includes the **oc** client tooling for invoking command line operations from within Jenkins jobs.
- It provides Dockerfiles for both Red Hat Enterprise Linux (RHEL) and **localdev** images.

Two more images that extend the base image are also provided:

- Maven v3.5 image
- Node.js v8 image

The Maven and Node.js Jenkins agent images provide Dockerfiles for the Universal Base Image (UBI) that you can reference when building new agent images. Also note the **contrib** and **contrib/bin** subdirectories. They allow for the insertion of configuration files and executable scripts for your image.



IMPORTANT

Use and extend an appropriate agent image version for the your of OpenShift Container Platform. If the **oc** client version that is embedded in the agent image is not compatible with the OpenShift Container Platform version, unexpected behavior can result.

8.3.1. Jenkins agent images

The OpenShift Container Platform Jenkins agent images are available on **quay.io** or **registry.redhat.io**.

Jenkins images are available through the Red Hat Registry:

```
$ docker pull registry.redhat.io/openshift4/ose-jenkins:<v4.1.4>
$ docker pull registry.redhat.io/openshift4/ose-jenkins-agent-nodejs:<v4.1.4>
$ docker pull registry.redhat.io/openshift4/ose-jenkins-agent-maven:<v4.1.4>
$ docker pull registry.redhat.io/openshift4/ose-jenkins-agent-base:<v4.1.4>
```

To use these images, you can either access them directly from **quay.io** or **registry.redhat.io** or push them into your OpenShift Container Platform container image registry.

8.3.2. Jenkins agent environment variables

Each Jenkins agent container can be configured with the following environment variables.

Variable	Definition	Example values and settings
JAVA_MAX_HEAP_PARAM, CONTAINER_HEAP_PERCENT, JENKINS_MAX_HEAP_UPPER_BOUND_MB	<p>These values control the maximum heap size of the Jenkins JVM. If JAVA_MAX_HEAP_PARAM is set, its value takes precedence. Otherwise, the maximum heap size is dynamically calculated as CONTAINER_HEAP_PERCENT of the container memory limit, optionally capped at JENKINS_MAX_HEAP_UPPER_BOUND_MB MiB.</p> <p>By default, the maximum heap size of the Jenkins JVM is set to 50% of the container memory limit with no cap.</p>	<p>JAVA_MAX_HEAP_PARAM example setting: -Xmx512m</p> <p>CONTAINER_HEAP_PERCENT default: 0.5, or 50%</p> <p>JENKINS_MAX_HEAP_UPPER_BOUND_MB example setting: 512 MiB</p>

Variable	Definition	Example values and settings
JAVA_INITIAL_HEAP_PARAMETER, CONTAINER_INITIAL_PERCENT	<p>These values control the initial heap size of the Jenkins JVM. If JAVA_INITIAL_HEAP_PARAMETER is set, its value takes precedence. Otherwise, the initial heap size is dynamically calculated as CONTAINER_INITIAL_PERCENT of the dynamically calculated maximum heap size.</p> <p>By default, the JVM sets the initial heap size.</p>	<p>JAVA_INITIAL_HEAP_PARAMETER example setting: -Xms32m</p> <p>CONTAINER_INITIAL_PERCENT example setting: 0.1, or 10%</p>
CONTAINER_CORE_LIMIT	If set, specifies an integer number of cores used for sizing numbers of internal JVM threads.	Example setting: 2
JAVA_TOOL_OPTIONS	Specifies options to apply to all JVMs running in this container. It is not recommended to override this value.	<p>Default: -</p> <p>XX:+UnlockExperimentalVMOptions -</p> <p>XX:+UseCGroupMemoryLimitForHeap -</p> <p>Dsun.zip.disableMemoryMapping=true</p>
JAVA_GC_OPTS	Specifies Jenkins JVM garbage collection parameters. It is not recommended to override this value.	<p>Default: -</p> <p>XX:+UseParallelGC -</p> <p>XX:MinHeapFreeRatio=5 -</p> <p>XX:MaxHeapFreeRatio=10 -</p> <p>XX:GCTimeRatio=4 -</p> <p>XX:AdaptiveSizePolicyWeight=90</p>
JENKINS_JAVA_OVERRIDES	Specifies additional options for the Jenkins JVM. These options are appended to all other options, including the Java options above, and can be used to override any of them, if necessary. Separate each additional option with a space; if any option contains space characters, escape them with a backslash.	<p>Example settings: -Dfoo -Dbar; -</p> <p>Dfoo=first\ value -</p> <p>Dbar=second\ value</p>

8.3.3. Jenkins agent memory requirements

A JVM is used in all Jenkins agents to host the Jenkins JNLP agent as well as to run any Java applications such as **javac**, Maven, or Gradle.

For memory efficiency, the Jenkins image dynamically uses a 32-bit JVM if it runs in a container with a memory limit under 2 GiB by default. The JVM choice applies by default both for the Jenkins JNLP agent as well as for any other Java processes within the agent container.

By default, the Jenkins JNLP agent JVM uses 50% of the container memory limit for its heap. This value can be modified by the **CONTAINER_HEAP_PERCENT** environment variable. It can also be capped at an upper limit or overridden entirely.

By default any other processes run in the Jenkins agent container, such as shell scripts or **oc** commands run from pipelines, cannot use more than the remaining 50% memory limit without provoking an OOM kill.

By default, each further JVM process that runs in a Jenkins agent container uses up to 25% of the container memory limit for its heap. It might be necessary to tune this limit for many build workloads.

8.3.4. Jenkins agent Gradle builds

Hosting Gradle builds in the Jenkins agent on OpenShift Container Platform presents additional complications because in addition to the Jenkins JNLP agent and Gradle JVMs, Gradle spawns a third JVM to run tests if they are specified.

The following settings are suggested as a starting point for running Gradle builds in a memory constrained Jenkins agent on OpenShift Container Platform. You can modify these settings as required.

- Ensure the long-lived Gradle daemon is disabled by adding **org.gradle.daemon=false** to the **gradle.properties** file.
- Disable parallel build execution by ensuring **org.gradle.parallel=true** is not set in the **gradle.properties** file and that **--parallel** is not set as a command line argument.
- To prevent Java compilations running out-of-process, set **java { options.fork = false }** in the **build.gradle** file .
- Disable multiple additional test processes by ensuring **test { maxParallelForks = 1 }** is set in the **build.gradle** file.
- Override the Gradle JVM memory parameters by the **GRADLE_OPTS**, **JAVA_OPTS** or **JAVA_TOOL_OPTIONS** environment variables.
- Set the maximum heap size and JVM arguments for any Gradle test JVM by defining the **maxHeapSize** and **jvmArgs** settings in **build.gradle**, or through the **-Dorg.gradle.jvmargs** command line argument.

8.3.5. Jenkins agent pod retention

Jenkins agent pods, also known as slave pods, are deleted by default after the build completes or is stopped. This behavior can be changed by the Kubernetes plug-in *Pod Retention* setting. Pod retention can be set for all Jenkins builds, with overrides for each pod template. The following behaviors are supported:

- **Always** keeps the build pod regardless of build result.
- **Default** uses the plug-in value (pod template only).
- **Never** always deletes the pod.
- **On Failure** keeps the pod if it fails during the build.

You can override pod retention in the pipeline Jenkinsfile:

```
podTemplate(label: "mypod",  
  cloud: "openshift",  
  inheritFrom: "maven",  
  podRetention: onFailure(), 1  
  containers: [  
    ...  
  ]) {  
    node("mypod") {  
      ...  
    }  
  }  
}
```

1 Allowed values for **podRetention** are **never()**, **onFailure()**, **always()**, and **default()**.



WARNING

Pods that are kept might continue to run and count against resource quotas.