OpenShift Dedicated 4

Images

Creating and managing images and imagestreams in OpenShift Dedicated 4

Last Updated: 2020-06-18

OpenShift Dedicated 4 Images

Creating and managing images and imagestreams in OpenShift Dedicated 4

Legal Notice

Abstract

This document provides instructions for creating and managing images and imagestreams in OpenShift Dedicated 4. It also provides instructions on using templates.

Table of Contents

CHAPTER 1. UNDERSTANDING CONTAINERS, IMAGES, AND IMAGESTREAMS 4
1.1. IMAGES 4
1.2. CONTAINERS 4
1.3. IMAGE REGISTRY 5
1.4. IMAGE REPOSITORY 5
1.5. IMAGE TAGS 5
1.6. IMAGE IDS 5
1.7. USING IMAGESTREAMS 5
1.8. IMAGESTREAMTAGS 6
1.9. IMAGESTREAM IMAGES 6
1.10. IMAGESTREAM TRIGGERS 7
1.11. ADDITIONAL RESOURCES 7

CHAPTER 2. CREATING IMAGES 8
2.1. LEARNING CONTAINER BEST PRACTICES 8
2.1.1. General container image guidelines 8
Reuse images 8
Maintain compatibility within tags 8
Avoid multiple processes 8
Use exec in wrapper scripts 8
Clean temporary files 9
Place instructions in the proper order 9
Mark important ports 10
Set environment variables 10
Avoid default passwords 10
Avoid sshd 10
Use volumes for persistent data 10
2.1.2. OpenShift Dedicated-specific guidelines 11
Enable images for source-to-image (S2I) 11
Support arbitrary user ids 11
Use services for inter-image communication 12
Provide common libraries 12
Use environment variables for configuration 12
Set image metadata 13
Clustering 13
Logging 13
Liveness and readiness probes 13
Templates 13
2.2. INCLUDING METADATA IN IMAGES 14
2.2.1. Defining image metadata 14
2.3. TESTING S2I IMAGES 15
2.3.1. Understanding testing requirements 15
2.3.2. Generating scripts and tools 16
2.3.3. Testing locally 16
2.3.4. Basic testing workflow 16
2.3.5. Using OpenShift Dedicated for building the image 17

CHAPTER 3. MANAGING IMAGES 18
3.1. MANAGING IMAGES OVERVIEW 18
3.1.1. Images overview 18
3.2. TAGGING IMAGES 18

3.2.1. Image tags 18
3.2.2. Image tag conventions 18
3.2.3. Adding tags to imagestreams 19
3.2.4. Removing tags from imagestreams 20
3.2.5. Referencing images in imagestreams 20
3.2.6. Additional information 21
3.3. IMAGE PULL POLICY 21
3.3.1. Image pull policy overview 21
3.4. USING IMAGE PULL SECRETS 22
3.4.1. Allowing Pods to reference images across projects 22
3.4.2. Allowing Pods to reference images from other secured registries 22
3.4.2.1. Pulling from private registries with delegated authentication 23
3.4.3. Updating the global cluster pull secret 24

CHAPTER 4. USING TEMPLATES 25
4.1. UNDERSTANDING TEMPLATES 25
4.2. UPLOADING A TEMPLATE 25
4.3. CREATING AN APPLICATION BY USING THE WEB CONSOLE 25
4.4. CREATING OBJECTS FROM TEMPLATES BY USING THE CLI 26
4.4.1. Adding labels 26
4.4.2. Listing parameters 26
4.4.3. Generating a list of objects 27
4.5. MODIFYING UPLOADED TEMPLATES 28
4.6. USING INSTANT APP AND QUICKSTART TEMPLATES 28
4.6.1. Quickstart templates 29
4.6.1.1. Web framework Quickstart templates 29
4.7. WRITING TEMPLATES 30
4.7.1. Writing the template description 30
4.7.2. Writing template labels 32
4.7.3. Writing template parameters 32
4.7.4. Writing the template object list 34
4.7.5. Marking a template as bindable 35
4.7.6. Exposing template object fields 35
4.7.7. Waiting for template readiness 37
4.7.8. Creating a template from existing objects 39

CHAPTER 5. USING RUBY ON RAILS 40
5.1. SETTING UP THE DATABASE 40
5.2. WRITING YOUR APPLICATION 41
5.2.1. Creating a welcome page 42
5.2.2. Configuring application for OpenShift Dedicated 42
5.2.3. Storing your application in Git 43
5.3. DEPLOYING YOUR APPLICATION TO OPENSHIFT DEDICATED 44
5.3.1. Creating the database service 44
5.3.2. Creating the frontend service 45
5.3.3. Creating a route for your application 46

CHAPTER 1. 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.

1.1. IMAGES

Containers in OpenShift Dedicated 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 Dedicated 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 Dedicated 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**).

1.2. CONTAINERS

The basic units of OpenShift Dedicated 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 Dedicated 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 Dedicated, understanding their capabilities and terminology is important for understanding their role in OpenShift Dedicated 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 Dedicated.

1.3. IMAGE REGISTRY

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

```
registry.redhat.io
```

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

1.4. IMAGE REPOSITORY

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

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

1.5. IMAGE TAGS

An image tag is a label applied to a container image in a repository that distinguishes a specific image from other images in an 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.

OpenShift Dedicated provides the **oc tag** command, which is similar to the **docker tag** command, but operates on imagestreams instead of directly on images.

1.6. IMAGE IDS

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

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

1.7. USING IMAGESTREAMS

An imagestream and its associated tags provide an abstraction for referencing container images from within OpenShift Dedicated. 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 Dedicated's integrated registry.
- An external registry, for example **registry.redhat.io** or **hub.docker.com**.
- Other imagestreams in the OpenShift Dedicated 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 Dedicated 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 Dedicated 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.

1.8. IMAGESTREAMTAGS

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

1.9. 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.

1.10. 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.

1.11. ADDITIONAL RESOURCES

CHAPTER 2. CREATING IMAGES

Learn how to create your own container images, based on pre-built images that are ready to help you. The process includes learning best practices for writing images, defining metadata for images, testing images, and using a custom builder workflow to create images to use with OpenShift Dedicated.

2.1. LEARNING CONTAINER BEST PRACTICES

When creating container images to run on OpenShift Dedicated 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 Dedicated.

2.1.1. 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 Dedicated.

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 Dedicated 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 Dedicated 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 Dedicated 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 Dedicated.

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

2.1.2. OpenShift Dedicated-specific guidelines

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

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 Dedicated 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.

WARNING

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

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

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

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 Dedicated 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 Dedicated 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 Dedicated better consume your container images, allowing OpenShift Dedicated 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 Dedicated. 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 Dedicated 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

2.2. INCLUDING METADATA IN IMAGES

Defining image metadata helps OpenShift Dedicated better consume your container images, allowing OpenShift Dedicated 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.

2.2.1. 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 Dedicated 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 2.1. Supported Metadata

Variable	Description
io.openshift.tags	This label contains a list of tags represented as a list of comma-separated string values. The tags are the way to categorize the container images into broad areas of functionality. Tags help UI and generation tools to suggest relevant container images during the application creation process. `LABEL io.openshift.tags mongodb,mongodb24,nosql`
io.openshift.wants	Specifies a list of tags that the generation tools and the UI might use to provide relevant suggestions if you do not have the container images with specified tags already. For example, if the container image wants **mysql** and **redis** and you do not have the container image with **redis** tag, then UI might suggest you to add this image into your deployment. `LABEL io.openshift.wants mongodb,redis`

Variable	Description
io.k8s.description	This label can be used to give the container image consumers more detailed information about the service or functionality this image provides. The UI can then use this description together with the container image name to provide more human friendly information to end users. `LABEL io.k8s.description The MySQL 5.5 Server with master-slave replication support`
io.openshift.non-scalable	An image 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. `LABEL io.openshift.non-scalable true`

io.openshift.min-memory and **io.openshift.min-cpu**	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.
	
	`LABEL io.openshift.min-memory 8Gi`
	`LABEL io.openshift.min-cpu 4`

2.3. TESTING S2I IMAGES

As an Source-to-Image (S2I) builder image author, you can test your S2I image locally and use the OpenShift Dedicated 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.

2.3.1. Understanding testing requirements

The standard location for the ***test*** script is ***test/run***. This script is invoked by the OpenShift Dedicated 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.

2.3.2. 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.

2.3.3. 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
```

2.3.4. 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.

2.3.5. Using OpenShift Dedicated 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 Dedicated to build and push the image. Simply define a Docker build that points to your repository.

If your OpenShift Dedicated 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 3. MANAGING IMAGES

3.1. MANAGING IMAGES OVERVIEW

With OpenShift Dedicated you can interact with images and set up imagestreams, depending on where the images' registries are located, any authentication requirements around those registries, and how you want your builds and deployments to behave.

3.1.1. Images overview

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

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

3.2. TAGGING IMAGES

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

3.2.1. Image tags

An image tag is a label applied to a container image in a repository that distinguishes a specific image from other images in an 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.

OpenShift Dedicated provides the **oc tag** command, which is similar to the **docker tag** command, but operates on imagestreams instead of directly on images.

3.2.2. Image tag conventions

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

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

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

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

Table 3.1. Image tag naming conventions

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

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

3.2.3. Adding tags to imagestreams

An imagestream in OpenShift Dedicated comprises zero or more container images identified by tags.

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

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

Procedure

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

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

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

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

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

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

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

NOTE

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

- You can also add the **--scheduled=true** flag to have the destination tag be refreshed, or re-imported, periodically. The period is configured globally at the system level.

- The **--reference** flag creates an imagestreamtag that is not imported. The tag points to the source location, permanently.
  If you want to instruct OpenShift to always fetch the tagged image from the integrated registry, use **--reference-policy=local**. The registry uses the pull-through feature to serve the image to the client. By default, the image blobs are mirrored locally by the registry. As a result, they can be pulled more quickly the next time they are needed. The flag also allows for pulling from insecure registries without a need to supply **--insecure-registry** to the container runtime as long as the imagestream has an insecure annotation or the tag has an insecure import policy.

3.2.4. Removing tags from imagestreams

You can remove tags from an imagestream.

Procedure

To remove a tag completely from an imagestream run:

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

or:

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

3.2.5. Referencing images in imagestreams

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

Table 3.2. Imagestream reference types

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

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

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

Procedure

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

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

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

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

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

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

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

NOTE

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

You can also reference a third-party registry:

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

Or an image with a digest:

```
centos/ruby-22-centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b28e
```

3.2.6. Additional information

- Example imagestream definitions for CentOS imagestreams.

3.3. IMAGE PULL POLICY

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

3.3.1. Image pull policy overview

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

Table 3.3. imagePullPolicy values

Value	Description
Always	Always pull the image.

Value	Description
IfNotPresent	Only pull the image if it does not already exist on the node.
Never	Never pull the image.

If a container's **imagePullPolicy** parameter is not specified, OpenShift Dedicated sets it based on the image's tag:

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

3.4. USING IMAGE PULL SECRETS

If you are using OpenShift Dedicated's internal registry and are pulling from imagestreams located in the same project, then your Pod's service account should already have the correct permissions and no additional action should be required.

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

3.4.1. Allowing Pods to reference images across projects

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

Procedure

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

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

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

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

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

3.4.2. Allowing Pods to reference images from other secured registries

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

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

Procedure

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

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

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

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

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

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

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

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

- To use a secret for pushing and pulling build images, the secret must be mountable inside of a Pod. You can do this by running:

```
$ oc secrets link builder <pull_secret_name>
```

3.4.2.1. Pulling from private registries with delegated authentication

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

Procedure

1. Create a secret for the delegated authentication server:

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

```
    redhat-connect-sso

secret/redhat-connect-sso
```

2. Create a secret for the private registry:

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

secret/private-registry
```

3.4.3. Updating the global cluster pull secret

You can update the global pull secret for your cluster.

WARNING

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

Prerequisites

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

Procedure

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

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

1 Provide the path to the new pull secret file.

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

CHAPTER 4. USING TEMPLATES

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

4.1. 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 Dedicated. 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.

4.2. 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.

4.3. CREATING AN APPLICATION BY USING THE WEB CONSOLE

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

Procedure

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

NOTE

Only 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/rhscl/ruby-26-rhel7"
  tags:
    -
      name: "2.6"
      annotations:
        description: "Build and run Ruby 2.6 applications"
        iconClass: "icon-ruby"
        tags: "builder,ruby" 1
        supports: "ruby:2.6,ruby"
        version: "2.6"
```

1 Including **builder** here ensures this **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.

4.4. CREATING OBJECTS FROM TEMPLATES BY USING THE CLI

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

4.4.1. 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
```

4.4.2. 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.

4.4.3. 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=-
   ```

4.5. 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>
```

4.6. USING INSTANT APP AND QUICKSTART TEMPLATES

OpenShift Dedicated 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.

4.6.1. Quickstart templates

A Quickstart is a basic example of an application running on OpenShift Dedicated. 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 Dedicated 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.

4.6.1.1. 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)

4.7. 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
```

4.7.1. 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 (1)
  annotations:
    openshift.io/display-name: "CakePHP MySQL Example (Ephemeral)" (2)
    description: >-
      An example CakePHP application with a MySQL database. For more information
      about using this template, including OpenShift considerations, see
      https://github.com/sclorg/cakephp-ex/blob/master/README.md.


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

1. The unique name of the template.
2. A brief, user-friendly name, which can be employed by user interfaces.
3. A description of the template. Include enough detail that 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.
4. Additional template description. This may be displayed by the service catalog, for example.
5. Tags to be associated with the template for searching and grouping. Add tags that 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.
6. 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.
7. The name of the person or organization providing the template.
8. A URL referencing further documentation for the template.
9. A URL where support can be obtained for the template.
10. An instructional message that is displayed when this template is instantiated. This field should inform the user how to use the newly created resources. Parameter substitution is performed on the message before being displayed so that generated credentials and other parameters can be included in the output. Include links to any next-steps documentation that users should follow.

4.7.2. 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" 1
  app: "${NAME}" 2
```

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

2 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.

4.7.3. 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: "${SOURCE_REPOSITORY_URL}" (1)
          ref: "${SOURCE_REPOSITORY_REF}"
        contextDir: "${CONTEXT_DIR}"
  - kind: DeploymentConfig
    apiVersion: v1
    metadata:
      name: frontend
    spec:
      replicas: "${{REPLICA_COUNT}}" (2)
parameters:
  - name: SOURCE_REPOSITORY_URL (3)
    displayName: Source Repository URL (4)
    description: The URL of the repository with your application source code (5)
    value: https://github.com/sclorg/cakephp-ex.git (6)
    required: true (7)
  - name: GITHUB_WEBHOOK_SECRET
```

```
    description: A secret string used to configure the GitHub webhook
    generate: expression 8
    from: "[a-zA-Z0-9]{40}" 9
  - name: 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. Do not 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.

4.7.4. Writing the template object list

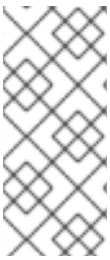
The main portion of the template is the list of objects which will be created when the template is instantiated. This can be any valid API object, such as a **BuildConfig**, **DeploymentConfig**, **Service**, etc. The object will be created exactly as defined here, with any parameter values substituted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

4.7.5. 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.

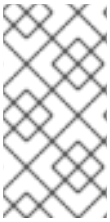
4.7.6. Exposing template object fields

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.

4.7.7. 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.

4.7.8. 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 5. USING RUBY ON RAILS

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

WARNING

Go through the whole tutorial to have an overview of all the steps necessary to run your application on the OpenShift Dedicated. 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 Dedicated 4.
- Make sure that an instance of OpenShift Dedicated 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.

5.1. 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.

5.2. 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-4.3.0
1 gem installed
```

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

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

3. Change into your new application directory:

```
$ cd rails-app
```

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

```
gem 'pg'
```

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

```
$ bundle install
```

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

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

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

```
$ rake db:create
```

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

5.2.1. 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 do not see the page, check the logs that are output to your server to debug.

5.2.2. Configuring application for OpenShift Dedicated

To have your application communicate with the PostgreSQL database service running in OpenShift Dedicated 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"] %>
```

5.2.3. Storing your application in Git

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

Prerequisites

- Install git.

Procedure

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

```
$ ls -1
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
```

-

5.3. DEPLOYING YOUR APPLICATION TO OPENSHIFT DEDICATED

You can deploy you application to OpenShift Dedicated.

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

Deploying your application in OpenShift Dedicated involves three steps:

- Creating a database service from OpenShift Dedicated's PostgreSQL image.
- Creating a frontend service from OpenShift Dedicated'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"
```

5.3.1. Creating the database service

Your Rails application expects a running database service. For this service use PostgeSQL 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
```

5.3.2. Creating the frontend service

To bring your application to OpenShift Dedicated, 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 Dedicated 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 Dedicated:

```
$ oc get pods
```

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

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 do not have to specify the **RAILS_ENV** environment variable.

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

5.3.3. 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 Dedicated 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.