



OpenShift Enterprise 3.2 Developer Guide

OpenShift Enterprise 3.2 Developer Reference

Red Hat OpenShift Documentation
Team

OpenShift Enterprise 3.2 Developer Guide

OpenShift Enterprise 3.2 Developer Reference

Legal Notice

Copyright © 2017 Red Hat, Inc.

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

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

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

Abstract

These topics help developers set up and configure a workstation to develop and deploy applications in an OpenShift Enterprise cloud environment with a command-line interface (CLI). This guide provides detailed instructions and examples to help developers: Monitor and browse projects with the web console Configure and utilize the CLI Generate configurations using templates Manage builds and webhooks Define and trigger deployments Integrate external services (databases, SaaS endpoints)

Table of Contents

CHAPTER 1. OVERVIEW	7
CHAPTER 2. APPLICATION LIFE CYCLE EXAMPLES	8
2.1. OVERVIEW	8
2.2. DEVELOPING ON OPENSIFT ENTERPRISE	8
2.3. DEVELOPING THEN DEPLOYING ON OPENSIFT ENTERPRISE	9
CHAPTER 3. AUTHENTICATION	10
3.1. WEB CONSOLE AUTHENTICATION	10
3.2. CLI AUTHENTICATION	10
CHAPTER 4. PROJECTS	12
4.1. OVERVIEW	12
4.2. CREATING A PROJECT	12
4.3. VIEWING PROJECTS	12
4.4. CHECKING PROJECT STATUS	13
4.5. FILTERING BY LABELS	13
4.6. DELETING A PROJECT	15
CHAPTER 5. CREATING NEW APPLICATIONS	16
5.1. OVERVIEW	16
5.2. CREATING AN APPLICATION USING THE CLI	16
5.3. CREATING AN APPLICATION USING THE WEB CONSOLE	22
CHAPTER 6. MIGRATING APPLICATIONS	26
6.1. OVERVIEW	26
6.2. MIGRATING DATABASE APPLICATIONS	27
6.3. MIGRATING WEB FRAMEWORK APPLICATIONS	34
6.4. QUICKSTART EXAMPLES	42
6.5. CONTINUOUS INTEGRATION AND DEPLOYMENT (CI/CD)	44
6.6. WEBHOOKS AND ACTION HOOKS	44
6.7. S2I TOOL	46
6.8. SUPPORT GUIDE	46
CHAPTER 7. APPLICATION TUTORIALS	51
7.1. OVERVIEW	51
7.2. QUICKSTART TEMPLATES	51
7.3. RUBY ON RAILS	52
CHAPTER 8. OPENING A REMOTE SHELL TO CONTAINERS	59
8.1. OVERVIEW	59
8.2. START A SECURE SHELL SESSION	59
8.3. SECURE SHELL SESSION HELP	59
CHAPTER 9. TEMPLATES	60
9.1. OVERVIEW	60
9.2. UPLOADING A TEMPLATE	60
9.3. CREATING FROM TEMPLATES USING THE WEB CONSOLE	60
9.4. CREATING FROM TEMPLATES USING THE CLI	63
9.5. MODIFYING AN UPLOADED TEMPLATE	65
9.6. USING THE INSTANT APP AND QUICKSTART TEMPLATES	65
9.7. WRITING TEMPLATES	66
CHAPTER 10. SERVICE ACCOUNTS	70

10.1. OVERVIEW	70
10.2. USER NAMES AND GROUPS	70
10.3. DEFAULT SERVICE ACCOUNTS AND ROLES	70
10.4. MANAGING SERVICE ACCOUNTS	71
10.5. MANAGING SERVICE ACCOUNT CREDENTIALS	71
10.6. MANAGING ALLOWED SECRETS	72
10.7. USING A SERVICE ACCOUNT'S CREDENTIALS INSIDE A CONTAINER	73
10.8. USING A SERVICE ACCOUNT'S CREDENTIALS EXTERNALLY	73
CHAPTER 11. BUILDS	75
11.1. OVERVIEW	75
11.2. DEFINING A BUILDCONFIG	75
11.3. SOURCE-TO-IMAGE STRATEGY OPTIONS	77
11.4. DOCKER STRATEGY OPTIONS	79
11.5. CUSTOM STRATEGY OPTIONS	81
11.6. BUILD INPUTS	83
11.7. GIT REPOSITORY SOURCE OPTIONS	84
11.8. DOCKERFILE SOURCE	89
11.9. BINARY SOURCE	90
11.10. IMAGE SOURCE	91
11.11. USING SECRETS DURING A BUILD	92
11.12. STARTING A BUILD	94
11.13. CANCELING A BUILD	95
11.14. DELETING A BUILDCONFIG	95
11.15. VIEWING BUILD DETAILS	96
11.16. ACCESSING BUILD LOGS	96
11.17. SETTING MAXIMUM DURATION	97
11.18. BUILD TRIGGERS	97
11.19. BUILD HOOKS	101
11.20. USING DOCKER CREDENTIALS FOR PUSHING AND PULLING IMAGES	103
11.21. BUILD RUN POLICY	105
11.22. BUILD OUTPUT	107
11.23. USING EXTERNAL ARTIFACTS DURING A BUILD	108
11.24. BUILD RESOURCES	109
11.25. TROUBLESHOOTING	110
CHAPTER 12. MANAGING IMAGES	112
12.1. OVERVIEW	112
12.2. TAGGING IMAGES	112
12.3. IMAGE PULL POLICY	116
12.4. ACCESSING THE INTERNAL REGISTRY	117
12.5. USING IMAGE PULL SECRETS	117
12.6. IMPORTING TAG AND IMAGE METADATA	119
CHAPTER 13. QUOTAS AND LIMIT RANGES	124
13.1. OVERVIEW	124
13.2. QUOTAS	124
13.3. LIMIT RANGES	131
13.4. COMPUTE RESOURCES	137
13.5. PROJECT RESOURCE LIMITS	140
CHAPTER 14. DEPLOYMENTS	141
14.1. OVERVIEW	141
14.2. CREATING A DEPLOYMENT CONFIGURATION	141
14.3. STARTING A DEPLOYMENT	142

14.3. STARTING A DEPLOYMENT	142
14.4. VIEWING A DEPLOYMENT	143
14.5. CANCELING A DEPLOYMENT	143
14.6. RETRYING A DEPLOYMENT	143
14.7. ROLLING BACK A DEPLOYMENT	144
14.8. EXECUTING COMMANDS INSIDE A CONTAINER	144
14.9. VIEWING DEPLOYMENT LOGS	145
14.10. TRIGGERS	145
14.11. STRATEGIES	147
14.12. LIFECYCLE HOOKS	150
14.13. DEPLOYMENT RESOURCES	152
14.14. MANUAL SCALING	153
14.15. ASSIGNING PODS TO SPECIFIC NODES	153
14.16. RUNNING A POD WITH A DIFFERENT SERVICE ACCOUNT	154
CHAPTER 15. ROUTES	155
15.1. OVERVIEW	155
15.2. CREATING ROUTES	155
CHAPTER 16. INTEGRATING EXTERNAL SERVICES	158
16.1. OVERVIEW	158
16.2. EXTERNAL MYSQL DATABASE	158
16.3. EXTERNAL SAAS PROVIDER	161
CHAPTER 17. SECRETS	165
17.1. OVERVIEW	165
17.2. PROPERTIES OF SECRETS	165
17.3. CREATING AND USING SECRETS	166
17.4. RESTRICTIONS	167
17.5. EXAMPLES	167
17.6. TROUBLESHOOTING	169
CHAPTER 18. CONFIGMAPS	170
18.1. OVERVIEW	170
18.2. CREATING CONFIGMAPS	171
18.3. USE CASES: CONSUMING CONFIGMAPS IN PODS	174
18.4. EXAMPLE: CONFIGURING REDIS	177
18.5. RESTRICTIONS	179
CHAPTER 19. USING DAEMONSETS	180
19.1. OVERVIEW	180
19.2. CREATING DAEMONSETS	180
CHAPTER 20. POD AUTOSCALING	182
20.1. OVERVIEW	182
20.2. REQUIREMENTS FOR USING HORIZONTAL POD AUTOSCALERS	182
20.3. SUPPORTED METRICS	182
20.4. AUTOSCALING	182
20.5. CREATING A HORIZONTAL POD AUTOSCALER	183
20.6. VIEWING A HORIZONTAL POD AUTOSCALER	184
CHAPTER 21. MANAGING VOLUMES	185
21.1. OVERVIEW	185
21.2. GENERAL CLI USAGE	185
21.3. ADDING VOLUMES	186
21.4. UPDATING VOLUMES	188

21.5. REMOVING VOLUMES	188
21.6. LISTING VOLUMES	189
CHAPTER 22. USING PERSISTENT VOLUMES	191
22.1. OVERVIEW	191
22.2. REQUESTING STORAGE	191
22.3. VOLUME AND CLAIM BINDING	191
22.4. CLAIMS AS VOLUMES IN PODS	192
22.5. VOLUME AND CLAIM PRE-BINDING	192
CHAPTER 23. EXECUTING REMOTE COMMANDS	195
23.1. OVERVIEW	195
23.2. BASIC USAGE	195
23.3. PROTOCOL	195
CHAPTER 24. COPYING FILES TO OR FROM A CONTAINER	197
24.1. OVERVIEW	197
24.2. BASIC USAGE	197
24.3. BACKING UP AND RESTORING DATABASES	197
24.4. REQUIREMENTS	198
24.5. SPECIFYING THE COPY SOURCE	199
24.6. SPECIFYING THE COPY DESTINATION	199
24.7. DELETING FILES AT THE DESTINATION	199
CHAPTER 25. PORT FORWARDING	200
25.1. OVERVIEW	200
25.2. BASIC USAGE	200
25.3. PROTOCOL	201
CHAPTER 26. SHARED MEMORY	202
26.1. OVERVIEW	202
26.2. POSIX SHARED MEMORY	202
CHAPTER 27. APPLICATION HEALTH	204
27.1. OVERVIEW	204
27.2. CONTAINER HEALTH CHECKS USING PROBES	204
CHAPTER 28. EVENTS	207
28.1. OVERVIEW	207
28.2. VIEWING EVENTS WITH THE CLI	207
28.3. VIEWING EVENTS IN THE CONSOLE	207
28.4. COMPREHENSIVE LIST OF EVENTS	207
CHAPTER 29. DOWNWARD API	211
29.1. OVERVIEW	211
29.2. SELECTING FIELDS	211
29.3. USING ENVIRONMENT VARIABLES	212
29.4. USING THE VOLUME PLUG-IN	213
CHAPTER 30. MANAGING ENVIRONMENT VARIABLES	215
30.1. SETTING AND UNSETTING ENVIRONMENT VARIABLES	215
30.2. LIST ENVIRONMENT VARIABLES	215
30.3. SET ENVIRONMENT VARIABLES	215
30.4. UNSET ENVIRONMENT VARIABLES	216
CHAPTER 31. JOBS	218

31.1. OVERVIEW	218
31.2. CREATING A JOB	218
31.3. SCALING A JOB	219
31.4. SETTING MAXIMUM DURATION	219
CHAPTER 32. REVISION HISTORY: DEVELOPER GUIDE	220
32.1. MON APR 03 2017	220
32.2. TUE MAR 14 2017	220
32.3. TUE FEB 21 2017	220
32.4. MON JAN 30 2017	220
32.5. MON JAN 16 2017	220
32.6. MON JAN 09 2017	221
32.7. TUE OCT 11 2016	221
32.8. TUE OCT 04 2016	221
32.9. TUE SEP 13 2016	221
32.10. TUE SEP 06 2016	221
32.11. MON AUG 29 2016	222
32.12. MON AUG 08 2016	222
32.13. MON AUG 01 2016	222
32.14. WED JUL 27 2016	223
32.15. THU JUL 14 2016	223
32.16. TUE JUN 14 2016	224
32.17. FRI JUN 10 2016	224
32.18. MON MAY 30 2016	224
32.19. THU MAY 12 2016	225

CHAPTER 1. OVERVIEW

This guide helps developers set up and configure a workstation to develop and deploy applications in an OpenShift Enterprise cloud environment with a command-line interface (CLI). This guide provides detailed instructions and examples to help developers:

- ✳ Monitor and browse projects with the web console.
- ✳ Configure and utilize the CLI.
- ✳ Generate configurations using templates.
- ✳ Manage builds and webhooks.
- ✳ Define and trigger deployments.
- ✳ Integrate external services (databases, SaaS endpoints).

CHAPTER 2. APPLICATION LIFE CYCLE EXAMPLES

2.1. OVERVIEW

As a PaaS, OpenShift Enterprise is designed for building and deploying applications. Depending on how much you want to involve OpenShift Enterprise in the development process, you can choose to develop on OpenShift Enterprise and use it to continuously develop an application, or you can deploy a fully developed application onto an OpenShift Enterprise instance.

2.2. DEVELOPING ON OPENSIFT ENTERPRISE



OPENSIFT_396538_0316

You can develop your application on OpenShift Enterprise directly. Use the following process if you plan to use OpenShift Enterprise as a method to build and deploy your application:

Initial Planning

- » What does your application do?
- » What programming language will it be developed in?

Access to OpenShift Enterprise

- » OpenShift Enterprise should be installed by this point, either by yourself or an administrator within your organization.

Develop

- » Using your editor/IDE of choice, create a basic skeleton of an application. It should be developed enough to tell OpenShift Enterprise [what kind of application it is](#).
- » Push the code to your Git repository.

Generate

- » [Create a basic application](#) using the **new-app** command. OpenShift Enterprise generates build and deployment configurations.

Manage

- » Start developing your application code.
- » Ensure your application builds successfully.
- » Continue to locally develop and polish your code.
- » Push your code to a Git repository.
- » Is any extra configuration needed? Explore the [Developer Guide](#) for more options.

Verify

- ✧ You can verify your application in a number of ways. You can push your changes to your application's Git repository, and use OpenShift Enterprise to rebuild and redeploy your application. Alternatively, you can hot deploy using **rsync** to synchronize your code changes into a running pod.

2.3. DEVELOPING THEN DEPLOYING ON OPENSIFT ENTERPRISE



OPENSIFT_396538_0316

Another possible application life cycle is to develop locally, then use OpenShift Enterprise to deploy your fully developed application. Use the following process if you plan to have application code already, then want to build and deploy onto an OpenShift Enterprise installation when completed:

Initial Planning

- ✧ What does your application do?
- ✧ What programming language will it be developed in?

Develop

- ✧ Develop your application code using your editor/IDE of choice.
- ✧ Build and test your application code locally.
- ✧ Push your code to a Git repository.

Access to OpenShift Enterprise

- ✧ OpenShift Enterprise should be installed by this point, either by yourself or an administrator within your organization.

Generate

- ✧ [Create a basic application](#) using the **new-app** command. OpenShift Enterprise generates build and deployment configurations.

Verify

- ✧ Ensure that the application that you have built and deployed in the above Generate step is successfully running on OpenShift Enterprise.

Manage

- ✧ Continue to develop your application code until you are happy with the results.
- ✧ Rebuild your application in OpenShift Enterprise to accept any newly pushed code.
- ✧ Is any extra configuration needed? Explore the [Developer Guide](#) for more options.

CHAPTER 3. AUTHENTICATION

3.1. WEB CONSOLE AUTHENTICATION

When accessing the [web console](#) from a browser at `<master_public_addr>:8443`, you are automatically redirected to a login page.

Review the [browser versions and operating systems](#) that can be used to access the web console.

You can provide your login credentials on this page to obtain a token to make API calls. After logging in, you can navigate your projects using the [web console](#).

3.2. CLI AUTHENTICATION

You can authenticate from the command line using the CLI command **oc login**. You can [get started with the CLI](#) by running this command without any options:

```
$ oc login
```

The command's interactive flow helps you establish a session to an OpenShift Enterprise server with the provided credentials. If any information required to successfully log in to an OpenShift Enterprise server is not provided, the command prompts for user input as required. The [configuration](#) is automatically saved and is then used for every subsequent command.

All configuration options for the **oc login** command, listed in the **oc login --help** command output, are optional. The following example shows usage with some common options:

```
$ oc login [-u=<username>] \
  [-p=<password>] \
  [-s=<server>] \
  [-n=<project>] \
  [--certificate-authority=</path/to/file.crt>|--insecure-skip-tls-verify]
```

The following table describes these common options:

Table 3.1. Common CLI Configuration Options

Option	Syntax	Description
-s, --server	<pre>\$ oc login -s=<server></pre>	Specifies the host name of the OpenShift Enterprise server. If a server is provided through this flag, the command does not ask for it interactively. This flag can also be used if you already have a CLI configuration file and want to log in and switch to another server.

Option	Syntax	Description
-u, --username and -p, --password	<pre>\$ oc login -u=<username> -p=<password></pre>	Allows you to specify the credentials to log in to the OpenShift Enterprise server. If user name or password are provided through these flags, the command does not ask for it interactively. These flags can also be used if you already have a configuration file with a session token established and want to log in and switch to another user name.
-n, --namespace	<pre>\$ oc login -u=<username> -p=<password> -n=<project></pre>	A global CLI option which, when used with oc login , allows you to specify the project to switch to when logging in as a given user.
--certificate-authority	<pre>\$ oc login --certificate-authority=<path/to/file.crt></pre>	Correctly and securely authenticates with an OpenShift Enterprise server that uses HTTPS. The path to a certificate authority file must be provided.
--insecure-skip-tls-verify	<pre>\$ oc login --insecure-skip-tls-verify</pre>	Allows interaction with an HTTPS server bypassing the server certificate checks; however, note that it is not secure. If you try to oc login to a HTTPS server that does not provide a valid certificate, and this or the --certificate-authority flags were not provided, oc login will prompt for user input to confirm (y/N kind of input) about connecting insecurely.

CLI configuration files allow you to easily [manage multiple CLI profiles](#).

Note

If you have access to administrator credentials but are no longer logged in as the [default system user **system:admin**](#), you can log back in as this user at any time as long as the credentials are still present in your [CLI configuration file](#). The following command logs in and switches to the **default** project:

```
$ oc login -u system:admin -n default
```

CHAPTER 4. PROJECTS

4.1. OVERVIEW

A [project](#) allows a community of users to organize and manage their content in isolation from other communities.

4.2. CREATING A PROJECT

If [allowed](#) by your cluster administrator, you can create a new project using [the CLI](#) or the [web console](#).

To create a new project using the CLI:

```
$ oc new-project <project_name> \
  --description="<description>" --display-name="<display_name>"
```

For example:

```
$ oc new-project hello-openshift \
  --description="This is an example project to demonstrate OpenShift
v3" \
  --display-name="Hello OpenShift"
```



Note

The number of projects you are allowed to create [may be limited by the system administrator](#). Once your limit is reached, you may need to delete an existing project in order to create a new one.

4.3. VIEWING PROJECTS

When viewing projects, you are restricted to seeing only the projects you have access to view based on the [authorization policy](#).

To view a list of projects:

```
$ oc get projects
```

You can change from the current project to a different project for CLI operations. The specified project is then used in all subsequent operations that manipulate project-scoped content:

```
$ oc project <project_name>
```

You can also use the [web console](#) to view and change between projects. After [authenticating](#) and logging in, you are presented with a list of projects that you have access to:



If you use [the CLI](#) to [create a new project](#), you can then refresh the page in the browser to see the new project.

Selecting a project brings you to the [project overview](#) for that project.

4.4. CHECKING PROJECT STATUS

The **oc status** command provides a high-level overview of the current project, with its components and their relationships. This command takes no argument:

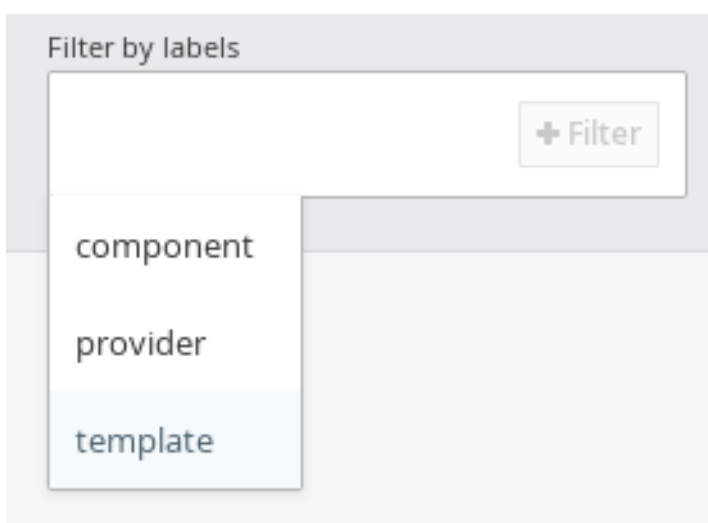
```
$ oc status
```

4.5. FILTERING BY LABELS

You can filter the contents of a project page in the [web console](#) by using the [labels](#) of a resource. You can pick from a suggested label name and values, or type in your own. Multiple filters can be added. When multiple filters are applied, resources must match all of the filters to remain visible.

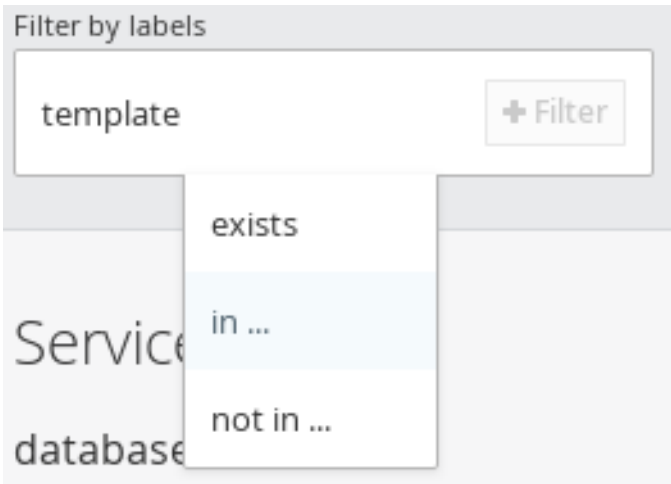
To filter by labels:

1. Select a label type:

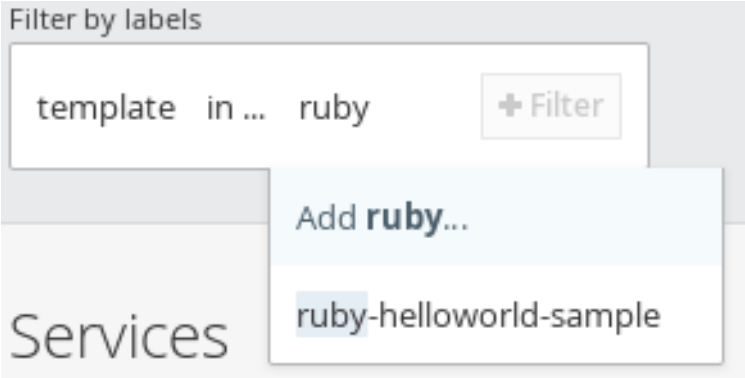


2. Select one of the following:

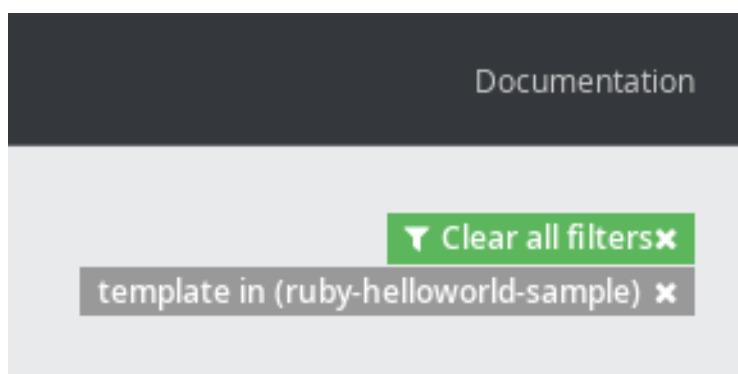
exists	Verify that the label name exists, but ignore its value.
in	Verify that the label name exists and is equal to one of the selected values.
not in	Verify that the label name does not exist, or is not equal to any of the selected values.



a. If you selected **in** or **not in**, select a set of values then select **Filter**:



- After adding filters, you can stop filtering by selecting **Clear all filters** or by clicking individual filters to remove them:



4.6. DELETING A PROJECT

When you delete a project, the server updates the project status to Terminating from Active. The server then clears all content from a project that is Terminating before finally removing the project. While a project is in Terminating status, a user cannot add new content to the project. Projects can be deleted from the CLI or the web console.

To delete a project using the CLI:

```
$ oc delete project <project_name>
```

CHAPTER 5. CREATING NEW APPLICATIONS

5.1. OVERVIEW

You can create a new OpenShift Enterprise application from source code, images, or templates by using either the OpenShift CLI or web console.

5.2. CREATING AN APPLICATION USING THE CLI

5.2.1. Creating an Application From Source Code

The **new-app** command allows you to create applications using source code in a local or remote Git repository.

To create an application using a Git repository in a local directory:

```
$ oc new-app /path/to/source/code
```



Note

If using a local Git repository, the repository must have an **origin** remote that points to a URL accessible by the OpenShift Enterprise cluster.

You can use a subdirectory of your source code repository by specifying a **--context-dir** flag. To create an application using a remote Git repository and a context subdirectory:

```
$ oc new-app https://github.com/openshift/sti-ruby.git \
  --context-dir=2.0/test/puma-test-app
```

Also, when specifying a remote URL, you can specify a Git branch to use by appending #**<branch_name>** to the end of the URL:

```
$ oc new-app https://github.com/openshift/ruby-hello-world.git#beta4
```

Using **new-app** results in a [build configuration](#), which creates a new application [image](#) from your source code. It also constructs a [deployment configuration](#) to deploy the new image, and a [service](#) to provide load-balanced access to the deployment running your image.

OpenShift Enterprise automatically [detects](#) whether the **Docker** or **Sourcebuild strategy** is being used, and in the case of **Source** builds, [detects an appropriate language builder image](#).

Build Strategy Detection

If a **Dockerfile** is in the repository when creating a new application, OpenShift Enterprise generates a [Docker build strategy](#). Otherwise, it generates a [Source strategy](#).

You can specify a strategy by setting the **--strategy** flag to either **source** or **docker**.

■

```
$ oc new-app /home/user/code/myapp --strategy=docker
```

Language Detection

If creating a **Source** build, **new-app** attempts to determine the language builder to use by the presence of certain files in the root of the repository:

Table 5.1. Languages Detected by new-app

Language	Files
ruby	<i>Rakefile, Gemfile, config.ru</i>
jee	<i>pom.xml</i>
nodejs	<i>app.json, package.json</i>
php	<i>index.php, composer.json</i>
python	<i>requirements.txt, setup.py</i>
perl	<i>index.pl, cpanfile</i>

After a language is detected, **new-app** searches the OpenShift Enterprise server for [image stream](#) tags that have a **supports** annotation matching the detected language, or an image stream that matches the name of the detected language. If a match is not found, **new-app** searches the [Docker Hub registry](#) for an image that matches the detected language based on name.

You can override the image the builder uses for a particular source repository by specifying the image (either an image stream or container specification) and the repository, with a ~ as a separator.

For example, to use the **myproject/my-ruby** image stream with the source in a remote repository:

```
$ oc new-app myproject/my-ruby~https://github.com/openshift/ruby-hello-world.git
```

To use the **openshift/ruby-20-centos7:latest** container image stream with the source in a local repository:

```
$ oc new-app openshift/ruby-20-centos7:latest~/home/user/code/my-ruby-app
```

5.2.2. Creating an Application From an Image

You can deploy an application from an existing image. Images can come from image streams in the OpenShift Enterprise server, images in a specific registry or [Docker Hub registry](#), or images in the local Docker server.

The **new-app** command attempts to determine the type of image specified in the arguments passed to it. However, you can explicitly tell **new-app** whether the image is a Docker image (using the **--docker-image** argument) or an image stream (using the **-i** | **--image** argument).



Note

If you specify an image from your local Docker repository, you must ensure that the same image is available to the OpenShift Enterprise cluster nodes.

For example, to create an application from the DockerHub MySQL image:

```
$ oc new-app mysql
```

To create an application using an image in a private registry, specify the full Docker image specification:

```
$ oc new-app myregistry:5000/example/myimage
```



Note

If the registry containing the image is not [secured with SSL](#), cluster administrators must ensure that the Docker daemon on the OpenShift Enterprise node hosts is run with the **--insecure-registry** flag pointing to that registry. You must also tell **new-app** that the image comes from an insecure registry with the **--insecure-registry=true** flag.

You can create an application from an existing [image stream](#) and tag (optional) for the image stream:

```
$ oc new-app my-stream:v1
```

5.2.3. Creating an Application From a Template

You can create an application from a previously stored [template](#) or from a template file, by specifying the name of the template as an argument. For example, you can store a [sample application template](#) and use it to create an application.

To create an application from a stored template:

```
$ oc create -f examples/sample-app/application-template-stibuild.json
$ oc new-app ruby-helloworld-sample
```

To directly use a template in your local file system, without first storing it in OpenShift Enterprise, use the **-f** | **--file** argument:

```
$ oc new-app -f examples/sample-app/application-template-stibuild.json
```

Template Parameters

When creating an application based on a [template](#), use the **-p** | **--param** argument to set parameter values defined by the template:

```
$ oc new-app ruby-helloworld-sample \
  -p ADMIN_USERNAME=admin,ADMIN_PASSWORD=mypassword
```

5.2.4. Further Modifying Application Creation

The **new-app** command generates OpenShift Enterprise objects that will build, deploy, and run the application being created. Normally, these objects are created in the current project using names derived from the input source repositories or the input images. However, **new-app** allows you to modify this behavior.

The set of objects created by **new-app** depends on the artifacts passed as input: source repositories, images, or templates.

Table 5.2. new-app Output Objects

Object	Description
BuildConfig	A BuildConfig is created for each source repository specified in the command line. The BuildConfig specifies the strategy to use, the source location, and the build output location.
ImageStreams	For BuildConfig , two ImageStreams are usually created: one to represent the input image (the builder image in the case of Source builds or FROM image in case of Docker builds), and another one to represent the output image. If a container image was specified as input to new-app , then an image stream is created for that image as well.
DeploymentConfig	A DeploymentConfig is created either to deploy the output of a build, or a specified image. The new-app command creates EmptyDir volumes for all Docker volumes that are specified in containers included in the resulting DeploymentConfig .

Object	Description
Service	The new-app command attempts to detect exposed ports in input images. It uses the lowest numeric exposed port to generate a service that exposes that port. In order to expose a different port, after new-app has completed, simply use the oc expose command to generate additional services.
Other	Other objects can be generated when instantiating templates .

5.2.4.1. Specifying Environment Variables

When generating applications from a [source](#) or an [image](#), you can use the **-e** | **--env** argument to pass environment variables to the application container at run time:

```
$ oc new-app openshift/postgresql-92-centos7 \
  -e POSTGRESQL_USER=user \
  -e POSTGRESQL_DATABASE=db \
  -e POSTGRESQL_PASSWORD=password
```

5.2.4.2. Specifying Labels

When generating applications from [source](#), [images](#), or [templates](#), you can use the **-l** | **--label** argument to add labels to the created objects. Labels make it easy to collectively select, configure, and delete objects associated with the application.

```
$ oc new-app https://github.com/openshift/ruby-hello-world -l
name=hello-world
```

5.2.4.3. Viewing the Output Without Creation

To see a dry-run of what **new-app** will create, you can use the **-o** | **--output** argument with a **yaml** or **json** value. You can then use the output to preview the objects that will be created, or redirect it to a file that you can edit. Once you are satisfied, you can use **oc create** to create the OpenShift Enterprise objects.

To output **new-app** artifacts to a file, edit them, then create them:

```
$ oc new-app https://github.com/openshift/ruby-hello-world \
  -o yaml > myapp.yaml
$ vi myapp.yaml
$ oc create -f myapp.yaml
```

5.2.4.4. Creating Objects With Different Names

Objects created by **new-app** are normally named after the source repository, or the image used to generate them. You can set the name of the objects produced by adding a **--name** flag to the command:

```
$ oc new-app https://github.com/openshift/ruby-hello-world --name=myapp
```

5.2.4.5. Creating Objects in a Different Project

Normally, **new-app** creates objects in the current project. However, you can create objects in a different project that you have access to using the **-n** | **--namespace** argument:

```
$ oc new-app https://github.com/openshift/ruby-hello-world -n myproject
```

5.2.4.6. Creating Multiple Objects

The **new-app** command allows creating multiple applications specifying multiple parameters to **new-app**. Labels specified in the command line apply to all objects created by the single command. Environment variables apply to all components created from source or images.

To create an application from a source repository and a Docker Hub image:

```
$ oc new-app https://github.com/openshift/ruby-hello-world mysql
```



Note

If a source code repository and a builder image are specified as separate arguments, **new-app** uses the builder image as the builder for the source code repository. If this is not the intent, simply specify a specific builder image for the source using the **~** separator.

5.2.4.7. Grouping Images and Source in a Single Pod

The **new-app** command allows deploying multiple images together in a single pod. In order to specify which images to group together, use the **+** separator. The **--group** command line argument can also be used to specify the images that should be grouped together. To group the image built from a source repository with other images, specify its builder image in the group:

```
$ oc new-app nginx+mysql
```

To deploy an image built from source and an external image together:

```
$ oc new-app \
  ruby~https://github.com/openshift/ruby-hello-world \
  mysql \
  --group=ruby+mysql
```

5.2.4.8. Useful Edits

Following are some specific examples of useful [edits to make](#) in the *myapp.yaml* file.



Note

These examples presume *myapp.yaml* was created as a result of the **oc new-app ... -o yaml** command.

Example 5.1. Deploy to Selected Nodes

```
apiVersion: v1
items:
- apiVersion: v1
  kind: Project 1
  metadata:
    name: myapp
    annotations:
      openshift.io/node-selector: region=west 2
- apiVersion: v1
  kind: ImageStream
  ...
kind: List
metadata: {}
```

1

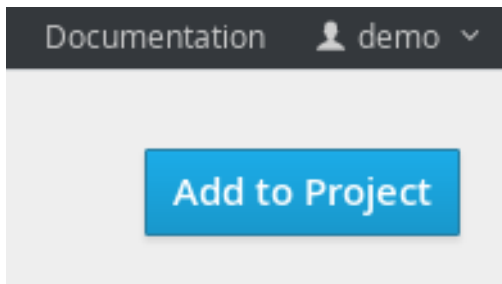
In *myapp.yaml*, the section that defines the **myapp** project has both **kind: Project** and **metadata.name: myapp**. If this section is missing, you should add it at the **second** level, as a new item of the list **items**, peer to the **kind: ImageStream** definitions.

2

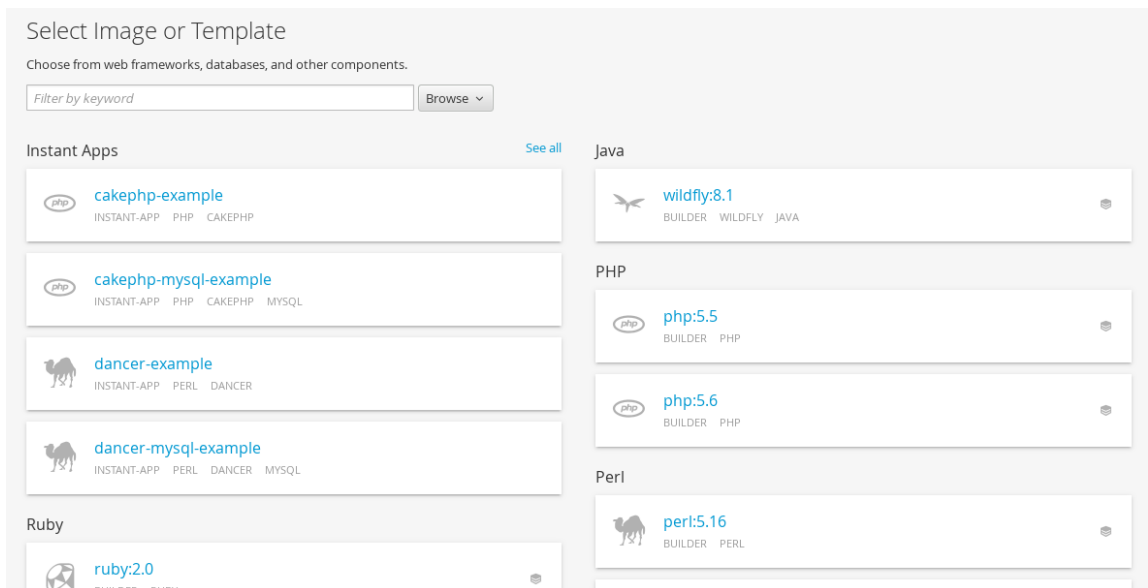
Add this [node selector](#) annotation to the **myapp** project to cause its pods to be deployed only on nodes that have the label **region=west**.

5.3. CREATING AN APPLICATION USING THE WEB CONSOLE

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 global library:



Note


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

```
kind: "ImageStream"
apiVersion: "v1"
metadata:
  name: "ruby"
  creationTimestamp: null
spec:
  dockerImageRepository:
    "registry.access.redhat.com/openshift3/ruby-20-rhel7"
  tags:
    -
      name: "2.0"
      annotations:
        description: "Build and run Ruby 2.0 applications"
        iconClass: "icon-ruby"
        tags: "builder,ruby" 1
        supports: "ruby:2.0,ruby"
        version: "2.0"
```

1

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

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



nodejs ¹

Version 0.10
Build and run NodeJS 0.10 applications

*** Name ²**

Used to uniquely identify within this project all the resources created to support the application.

*** Git Repository URL ³**

Sample repository for nodejs: <https://github.com/openshift/nodejs-ex.git> [Try it ↑](#)

Git Reference

Optional branch, tag, or commit.

Context Dir

Optional subdirectory for the application source code, used as the context directory for the build.

Routing ⁴

[About Routing](#)

☒ Create a route to the application ⓘ

Target port: 8080/TCP

Deployment Configuration ⁵

[About Deployment Configuration](#)

Autodeploy when

☒ New image is available

☒ Deployment configuration changes

Environment Variables ⓘ

Name	Value	Add
------	-------	-----

Build Configuration ⁶

[About Build Configuration](#)

☒ Configure a webhook build trigger ⓘ

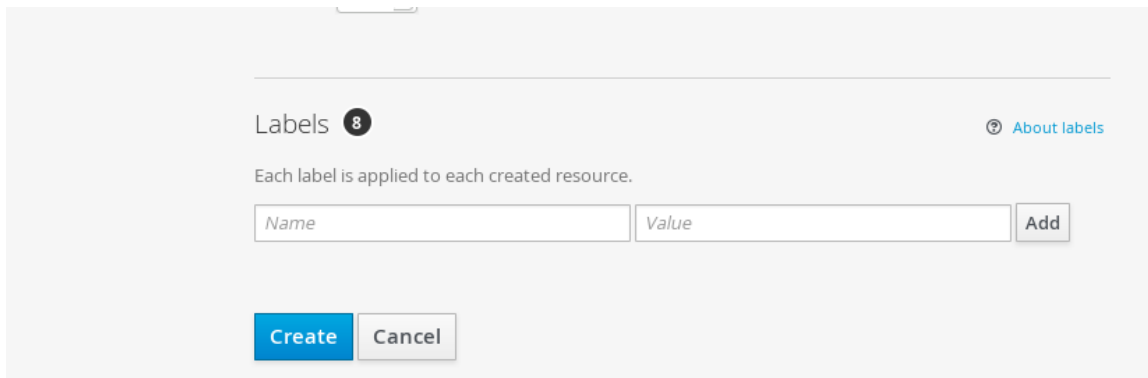
☒ Automatically build a new image when the builder image changes ⓘ

☒ Automatically build a new image when the build configuration changes

Scaling ⁷

[About Scaling](#)

Replicas: ↑ ↓



Labels **8** [? About labels](#)

Each label is applied to each created resource.

The builder image name and description.

The application name used for the generated OpenShift Enterprise objects.

The Git repository URL, reference, and context directory for your source code.

Routing configuration section for making this application publicly accessible.

Deployment configuration section for customizing [deployment triggers](#) and image environment variables.

Build configuration section for customizing [build triggers](#).

Replica [scaling](#) section for configuring the number of running instances of the application.

The [labels](#) to assign to all items generated for the application. You can add and edit labels for all objects here.



Note

To see all of the configuration options, click the "Show advanced build and deployment options" link.

CHAPTER 6. MIGRATING APPLICATIONS

6.1. OVERVIEW

This topic covers the migration procedure of OpenShift version 2 (v2) applications to OpenShift version 3 (v3).



Note

This topic uses some terminology that is specific to OpenShift v2. [Comparing OpenShift Enterprise 2 and OpenShift Enterprise 3](#) provides insight on the differences between the two versions and the language used.

To migrate OpenShift v2 applications to OpenShift Enterprise v3, all cartridges in the v2 application must be recorded as each v2 cartridge is equivalent with a corresponding image or template in OpenShift Enterprise v3 and they must be migrated individually. For each cartridge, all dependencies or required packages also must be recorded, as they must be included in the v3 images.

The general migration procedure is:

1. Back up the v2 application.

- ✦ Web cartridge: The source code can be backed up to a Git repository such as by pushing to a repository on GitHub.
- ✦ Database cartridge: The database can be backed up using a dump command (**mongodump**, **mysqldump**, **pg_dump**) to back up the database.
- ✦ Web and database cartridges: **rhc** client tool provides snapshot ability to back up multiple cartridges:

```
$ rhc snapshot save <app_name>
```

The snapshot is a tar file that can be unzipped, and its content is application source code and the database dump.

2. If the application has a database cartridge, create a v3 database application, sync the database dump to the pod of the new v3 database application, then restore the v2 database in the v3 database application with database restore commands.
3. For a web framework application, edit the application source code to make it v3 compatible. Then, add any dependencies or packages required in appropriate files in the Git repository. Convert v2 environment variables to corresponding v3 environment variables.
4. Create a v3 application from source (your Git repository) or from a quickstart with your Git URL. Also, add the database service parameters to the new application to link the database application to the web application.
5. In v2, there is an integrated Git environment and your applications automatically rebuild and restart whenever a change is pushed to your v2 Git repository. In v3, in order to have a build automatically triggered by source code changes pushed to your public Git repository, you must set up a [webhook](#) after the initial build in v3 is completed.

6.2. MIGRATING DATABASE APPLICATIONS

6.2.1. Overview

This topic reviews how to migrate MySQL, PostgreSQL, and MongoDB database applications from OpenShift version 2 (v2) to OpenShift version 3 (v3).

6.2.2. Supported Databases

v2	v3
MongoDB: 2.4	MongoDB: 2.4, 2.6
MySQL: 5.5	MySQL: 5.5, 5.6
PostgreSQL: 9.2	PostgreSQL: 9.2, 9.4

6.2.3. MySQL

1. Export all databases to a dump file and copy it to a local machine (into the current directory):

```
$ rhc ssh <v2_application_name>
$ mysqldump --skip-lock-tables -h $OPENSHIFT_MYSQL_DB_HOST -P
${OPENSHIFT_MYSQL_DB_PORT:-3306} -u
${OPENSHIFT_MYSQL_DB_USERNAME:-'admin'} \
--password="$OPENSHIFT_MYSQL_DB_PASSWORD" --all-databases >
~/app-root/data/all.sql
$ exit
```

2. Download **dbdump** to your local machine:

```
$ mkdir mysqldumpdir
$ rhc scp -a <v2_application_name> download mysqldumpdir app-
root/data/all.sql
```

3. Create a v3 **mysql-persistent** pod from template:

```
$ oc new-app mysql-persistent -p \
  MYSQL_USER=<your_v2_mysql_username> -p \
  MYSQL_PASSWORD=<your_v2_mysql_password> -p MYSQL_DATABASE=
<your_v2_database_name>
```

4. Check to see if the pod is ready to use:

```
$ oc get pods
```

5. When the pod is up and running, copy database archive files to your v3 MySQL pod:

```
$ oc rsync /local/mysqldumpdir
<mysql_pod_name>:/var/lib/mysql/data
```

6. Restore the database in the v3 running pod:

```
$ oc rsh <mysql_pod>
$ cd /var/lib/mysql/data/mysqldumpdir
```

In v3, to restore databases you need to access MySQL as **root** user.

In v2, the **\$OPENSIFT_MYSQL_DB_USERNAME** had full privileges on all databases. In v3, you must grant privileges to **\$MYSQL_USER** for each database.

```
$ mysql -u root
$ source all.sql
```

Grant all privileges on **<dbname>** to **<your_v2_username>@localhost**, then flush privileges.

7. Remove the dump directory from the pod:

```
$ cd ../; rm -rf /var/lib/mysql/data/mysqldumpdir
```

Supported MySQL Environment Variables

v2	v3
OPENSIFT_MYSQL_DB_HOST	[service_name]_SERVICE_HOST
OPENSIFT_MYSQL_DB_PORT	[service_name]_SERVICE_PORT
OPENSIFT_MYSQL_DB_USERNAME	MYSQL_USER
OPENSIFT_MYSQL_DB_PASSWORD	MYSQL_PASSWORD
OPENSIFT_MYSQL_DB_URL	
OPENSIFT_MYSQL_DB_LOG_DIR	
OPENSIFT_MYSQL_VERSION	

v2	v3
OPENSIFT_MYSQL_DIR	
OPENSIFT_MYSQL_DB_SOCKET	
OPENSIFT_MYSQL_IDENT	
OPENSIFT_MYSQL_AIO	MYSQL_AIO
OPENSIFT_MYSQL_MAX_ALLOWED_PACKET	MYSQL_MAX_ALLOWED_PACKET
OPENSIFT_MYSQL_TABLE_OPEN_CACHE	MYSQL_TABLE_OPEN_CACHE
OPENSIFT_MYSQL_SORT_BUFFER_SIZE	MYSQL_SORT_BUFFER_SIZE
OPENSIFT_MYSQL_LOWER_CASE_TABLE_NAMES	MYSQL_LOWER_CASE_TABLE_NAMES
OPENSIFT_MYSQL_MAX_CONNECTIONS	MYSQL_MAX_CONNECTIONS
OPENSIFT_MYSQL_FT_MIN_WORD_LEN	MYSQL_FT_MIN_WORD_LEN
OPENSIFT_MYSQL_FT_MAX_WORD_LEN	MYSQL_FT_MAX_WORD_LEN
OPENSIFT_MYSQL_DEFAULT_STORAGE_ENGINE	
OPENSIFT_MYSQL_TIMEZONE	
	MYSQL_DATABASE

v2	v3
	MYSQL_ROOT_PASSWORD
	MYSQL_MASTER_USER
	MYSQL_MASTER_PASSWORD

6.2.4. PostgreSQL

1. Back up the v2 PostgreSQL database from the gear:

```
$ rhc ssh -a <v2-application_name>
$ mkdir ~/app-root/data/tmp
$ pg_dump <database_name> | gzip > ~/app-
root/data/tmp/<database_name>.gz
```

2. Extract the backup file back to your local machine:

```
$ rhc scp -a <v2_application_name> download <local_dest> app-
root/data/tmp/<db-name>.gz
$ gzip -d <database-name>.gz
```



Note

Save the backup file to a separate folder for step 4.

3. Create the PostgreSQL service using the v2 application database name, user name and password to create the new service:

```
$ oc new-app postgresql-persistent -p POSTGRESQL_DATABASE=dbname
-p
POSTGRESQL_PASSWORD=password -p POSTGRESQL_USER=username
```

4. Check to see if the pod is ready to use:

```
$ oc get pods
```

5. When the pod is up and running, sync the backup directory to pod:

```
$ oc rsync /local/path/to/dir
<postgresql_pod_name>:/var/lib/pgsql/data
```

6. Remotely access the pod:

```
$ oc rsh <pod_name>
```

- 7. Restore the database:

```
psql dbname < /var/lib/pgsql/data/<database_backup_file>
```

- 8. Remove all backup files that are no longer needed:

```
$ rm /var/lib/pgsql/data/<database-backup-file>
```

Supported PostgreSQL Environment Variables

v2	v3
OPENSIFT_POSTGRESQL_DB_HOST	[service_name]_SERVICE_HOST
OPENSIFT_POSTGRESQL_DB_PORT	[service_name]_SERVICE_PORT
OPENSIFT_POSTGRESQL_DB_USERNAME	POSTGRESQL_USER
OPENSIFT_POSTGRESQL_DB_PASSWORD	POSTGRESQL_PASSWORD
OPENSIFT_POSTGRESQL_DB_LOG_DIR	
OPENSIFT_POSTGRESQL_DB_PID	
OPENSIFT_POSTGRESQL_DB_SOCKET_DIR	
OPENSIFT_POSTGRESQL_DB_URL	
OPENSIFT_POSTGRESQL_VERSION	
OPENSIFT_POSTGRESQL_SHARED_BUFFER S	
OPENSIFT_POSTGRESQL_MAX_CONNECTIONS	

v2	v3
OPENSIFT_POSTGRESQL_MAX_PREPARED_TRANSACTIONS	
OPENSIFT_POSTGRESQL_DATESTYLE	
OPENSIFT_POSTGRESQL_LOCALE	
OPENSIFT_POSTGRESQL_CONFIG	
OPENSIFT_POSTGRESQL_SSL_ENABLED	
	POSTGRESQL_DATABASE
	POSTGRESQL_ADMIN_PASSWORD

6.2.5. MongoDB



Note

- ✳ For OpenShift v3: MongoDB shell version 3.2.6
- ✳ For OpenShift v2: MongoDB shell version 2.4.9

1. Remotely access the v2 application via the **ssh** command:

```
$ rhc ssh <v2_application_name>
```

2. Run **mongodump**, specifying a single database with **-d <database_name> -c <collections>**. Without those options, dump all databases. Each database is dumped in its own directory:

```
$ mongodump -h $OPENSIFT_MONGODB_DB_HOST -o app-
root/repo/mydbdump -u 'admin' -p $OPENSIFT_MONGODB_DB_PASSWORD
$ cd app-root/repo/mydbdump/<database_name>; tar -cvzf
dbname.tar.gz
$ exit
```

3. Download **dbdump** to a local machine in the **mongodump** directory:

```
$ mkdir mongodump
$ rhc scp -a <v2 appname> download mongodump \
  app-root/repo/mydbdump/<dbname>/dbname.tar.gz
```

4. Start a MongoDB pod in v3. Because the latest image (3.2.6) does not include **mongo-tools**, to use **mongorestore** or **mongoimport** commands you need to edit the default **mongodb-persistent** template to specify the image tag that contains the **mongo-tools**, "**mongodb:2.4**". For that reason, the following **oc export** command and edit are necessary:

```
$ oc export template mongodb-persistent -n openshift -o json >
  mongodb-24persistent.json
```

Edit L80 of **mongodb-24persistent.json**; replace **mongodb:latest** with **mongodb:2.4**.

```
$ oc new-app --template=mongodb-persistent -n <project-name-that-
  template-was-created-in> \
  MONGODB_USER=user_from_v2_app -p \
  MONGODB_PASSWORD=password_from_v2_db -p \
  MONGODB_DATABASE=v2_dbname -p \
  MONGODB_ADMIN_PASSWORD=password_from_v2_db
$ oc get pods
```

5. When the mongodb pod is up and running, copy the database archive files to the v3 MongoDB pod:

```
$ oc rsync local/path/to/mongodump
  <mongodb_pod_name>:/var/lib/mongodb/data
$ oc rsh <mongodb_pod>
```

6. In the MongoDB pod, complete the following for each database you want to restore:

```
$ cd /var/lib/mongodb/data/mongodump
$ tar -xzf dbname.tar.gz
$ mongorestore -u $MONGODB_USER -p $MONGODB_PASSWORD -d dbname -v
  /var/lib/mongodb/data/mongodump
```

7. Check if the database is restored:

```
$ mongo admin -u $MONGODB_USER -p $MONGODB_ADMIN_PASSWORD
$ use dbname
$ show collections
$ exit
```

8. Remove the **mongodump** directory from the pod:

```
$ rm -rf /var/lib/mongodb/data/mongodump
```

Supported MongoDB Environment Variables

v2	v3
OPENSIFT_MONGODB_DB_HOST	[service_name]_SERVICE_HOST
OPENSIFT_MONGODB_DB_PORT	[service_name]_SERVICE_PORT
OPENSIFT_MONGODB_DB_USERNAME	MONGODB_USER
OPENSIFT_MONGODB_DB_PASSWORD	MONGODB_PASSWORD
OPENSIFT_MONGODB_DB_URL	
OPENSIFT_MONGODB_DB_LOG_DIR	
	MONGODB_DATABASE
	MONGODB_ADMIN_PASSWORD
	MONGODB_NOPREALLOC
	MONGODB_SMALLFILES
	MONGODB_QUIET
	MONGODB_REPLICA_NAME
	MONGODB_KEYFILE_VALUE

6.3. MIGRATING WEB FRAMEWORK APPLICATIONS

6.3.1. Overview

This topic reviews how to migrate Python, Ruby, PHP, Perl, Node.js, JBoss EAP, JBoss WS (Tomcat), and Wildfly 10 (JBoss AS) web framework applications from OpenShift version 2 (v2) to OpenShift version 3 (v3).

6.3.2. Python

1. Set up a new GitHub repository and add it as a remote branch to the current, local v2 Git repository:

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>.git
```

2. Push the local v2 source code to the new repository:

```
$ git push -u <remote-name> master
```

3. Ensure that all important files such as **setup.py**, **wsgi.py**, **requirements.txt**, and **etc** are pushed to new repository.

- ✳ Ensure all required packages for your application are included in **requirements.txt**.
- ✳ S2I-python does not support **mod_wsgi** anymore. However, **gunicorn** is supported and it is an alternative for **mod_wsgi**. So, add **gunicorn** package to **requirements.txt**.

4. Use the **oc** command to launch a new Python application from the builder image and source code:

```
$ oc new-app --strategy=source
python:3.3~https://github.com/<github-id>/<repo-name> --name=
<app-name> -e
<ENV_VAR_NAME>=<env_var_value>
```

Supported Python Versions

v2	v3
Python: 2.6, 2.7, 3.3	Python: 2.7, 3.3, 3.4
Django	Django-psql-example (quickstart)

6.3.3. Ruby

1. Set up a new GitHub repository and add it as a remote branch to the current, local v2 Git repository:

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>.git
```

2. Push the local v2 source code to the new repository:

```
$ git push -u <remote-name> master
```

3. If you do not have a Gemfile and are running a simple rack application, copy this Gemfile into the root of your source:

```
https://github.com/openshift/ruby-ex/blob/master/Gemfile
```



Note

The latest version of the **rack** gem that supports Ruby 2.0 is 1.6.4, so the Gemfile needs to be modified to **gem 'rack', "1.6.4"**.

For Ruby 2.2 or later, use the **rack** gem 2.0 or later.

4. Use the **oc** command to launch a new Ruby application from the builder image and source code:

```
$ oc new-app --strategy=source  
ruby:2.0~https://github.com/<github-id>/<repo-name>.git
```

Supported Ruby Versions

v2	v3
Ruby: 1.8, 1.9, 2.0	Ruby: 2.0, 2.2
Ruby on Rails: 3, 4	Rails-postgresql-example (quickstart)
Sinatra	

6.3.4. PHP

1. Set up a new GitHub repository and add it as a remote branch to the current, local v2 Git repository:

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. Push the local v2 source code to the new repository:

```
$ git push -u <remote-name> master
```


3. Use the **oc** command to launch a new PHP application from the builder image and source code:

```
$ oc new-app https://github.com/<github-id>/<repo-name>.git
--name=<app-name> -e <ENV_VAR_NAME>=<env_var_value>
```

Supported PHP Versions

v2	v3
PHP: 5.3, 5.4	PHP:5.5, 5.6
PHP 5.4 with Zend Server 6.1	
CodeIgniter 2	
HHVM	
Laravel 5.0	
	cakephp-mysql-example (quickstart)

6.3.5. Perl

1. Set up a new GitHub repository and add it as a remote branch to the current, local v2 Git repository:

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. Push the local v2 source code to the new repository:

```
$ git push -u <remote-name> master
```

3. Edit the local Git repository and push changes upstream to make it v3 compatible:

- a. In v2, CPAN modules reside in **.openshift/cpan.txt**. In v3, the s2i builder looks for a file named **cpanfile** in the root directory of the source.

```
$ cd <local-git-repository>
$ mv .openshift/cpan.txt cpanfile
```

Edit `cpanfile`, as it has a slightly different format:

format of <code>cpanfile</code>	format of <code>cpan.txt</code>
requires 'cpan::mod';	cpan::mod
requires 'Dancer';	Dancer
requires 'YAML';	YAML

- b. Remove **.openshift** directory



Note

In v3, **action_hooks** and **cron** tasks are not supported in the same way. See [Action Hooks](#) for more information.

4. Use the **oc** command to launch a new Perl application from the builder image and source code:

```
$ oc new-app https://github.com/<github-id>/<repo-name>.git
```

Supported Perl Versions

v2	v3
Perl: 5.10	Perl: 5.16, 5.20
	Dancer-mysql-example (quickstart)

6.3.6. Node.js

1. Set up a new GitHub repository and add it as a remote branch to the current, local Git repository:

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. Push the local v2 source code to the new repository:

```
$ git push -u <remote-name> master
```

3. Edit the local Git repository and push changes upstream to make it v3 compatible:

- a. Remove the **.openshift** directory.



Note

In v3, **action_hooks** and **cron** tasks are not supported in the same way. See [Action Hooks](#) for more information.

- b. Edit **server.js**.

- ✎ L116 server.js: 'self.app = express()';
- ✎ L25 server.js: self.ipaddress = '0.0.0.0';
- ✎ L26 server.js: self.port = 8080;



Note

Lines(L) are from the base V2 cartridge **server.js**.

4. Use the **oc** command to launch a new Node.js application from the builder image and source code:

```
$ oc new-app https://github.com/<github-id>/<repo-name>.git
--name=<app-name> -e <ENV_VAR_NAME>=<env_var_value>
```

Supported Node.js Versions

v2	v3
Node.js 0.10	Nodejs: 0.10
	Nodejs-mongodb-example (quickstart)

6.3.7. JBoss EAP

1. Set up a new GitHub repository and add it as a remote branch to the current, local Git repository:

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. Push the local v2 source code to the new repository:

```
$ git push -u <remote-name> master
```

-
- 3. If the repository includes pre-built **.war** files, they need to reside in the **deployments** directory off the root directory of the repository.
- 4. Create the new application using the JBoss EAP 6 builder image (jboss-eap64-openshift) and the source code repository from GitHub:

```
$ oc new-app --strategy=source jboss-eap64-openshift~https://github.com/<github-id>/<repo-name>.git
```

6.3.8. JBoss WS (Tomcat)

1. Set up a new GitHub repository and add it as a remote branch to the current, local Git repository:

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. Push the local v2 source code to the new repository:

```
$ git push -u <remote-name> master
```

3. If the repository includes pre-built **.war** files, they need to reside in the **deployments** directory off the root directory of the repository.
4. Create the new application using the JBoss Web Server 3 (Tomcat 7) builder image (jboss-webserver30-tomcat7) and the source code repository from GitHub:

```
$ oc new-app --strategy=source  
jboss-webserver30-tomcat7-openshift~https://github.com/<github-id>/<repo-name>.git  
--name=<app-name> -e <ENV_VAR_NAME>=<env_var_value>
```

6.3.9. JBoss AS (Wildfly 10)

1. Set up a new GitHub repository and add it as a remote branch to the current, local Git repository:

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. Push the local v2 source code to the new repository:

```
$ git push -u <remote-name> master
```

3. Edit the local Git repository and push the changes upstream to make it v3 compatible:
 - a. Remove **.openshift** directory.

**Note**

In v3, **action_hooks** and **cron** tasks are not supported in the same way. See [Action Hooks](#) for more information.

- b. Add the **deployments** directory to the root of the source repository. Move the **.war** files to 'deployments' directory.
4. Use the **oc** command to launch a new Wildfly application from the builder image and source code:

```
$ oc new-app https://github.com/<github-id>/<repo-name>.git
--image-stream="openshift/wildfly:10.0" --name=<app-name> -e
<ENV_VAR_NAME>=<env_var_value>
```

**Note**

The argument **--name** is optional to specify the name of your application. The argument **-e** is optional to add environment variables that are needed for build and deployment processes, such as **OPENSIFT_PYTHON_DIR**.

6.3.10. Supported JBoss/XPaas Versions

v2	v3
JBoss App Server 7	
Tomcat 6 (JBoss EWS 1.0)	jboss-webserver30-tomcat7-openshift: 1.1
Tomcat 7 (JBoss EWS 2.0)	
Vert.x 2.1	
WildFly App Server 10	
WildFly App Server 8.2.1.Final	
WildFly App Server 9	

v2	v3
CapeDwarf	
JBoss Data Virtualization 6	
JBoss Enterprise App Platform 6	jboss-eap64-openshift: 1.2, 1.3
JBoss Unified Push Server 1.0.0.Beta1, Beta2	
JBoss BPM Suite	
JBoss BRMS	
	jboss-eap70-openshift: 1.3-Beta
	eap64-https-s2i
	eap64-mongodb-persistent-s2i
	eap64-mysql-persistent-s2i
	eap64-psql-persistent-s2i

6.4. QUICKSTART EXAMPLES

6.4.1. Overview

Although there is no clear-cut migration path for v2 quickstart to v3 quickstart, the following quickstarts are currently available in v3. If you have an application with a database, rather than using **oc new-app** to create your application, then **oc new-app** again to start a separate database service and linking the two with common environment variables, you can use one of the following to instantiate the linked application and database at once, from your GitHub repository containing your source code. You can list all available templates with **oc get templates -n openshift**:

☞ CakePHP MySQL <https://github.com/openshift/cakephp-ex>

- template: cakephp-mysql-example
- ✳ Node.js MongoDB <https://github.com/openshift/nodejs-ex>
 - template: nodejs-mongodb-example
- ✳ Django PostgreSQL <https://github.com/openshift/django-ex>
 - template: django-psql-example
- ✳ Dancer MySQL <https://github.com/openshift/dancer-ex>
 - template: dancer-mysql-example
- ✳ Rails PostgreSQL <https://github.com/openshift/rails-ex>
 - template: rails-postgresql-example

6.4.2. Workflow

Run a **git clone** of one of the above template URLs locally. Add and commit your application source code and push a GitHub repository, then start a v3 quickstart application from one of the templates listed above:

1. Create a GitHub repository for your application.
2. Clone a quickstart template and add your GitHub repository as a remote:

```
$ git clone <one-of-the-template-URLs-listed-above>
$ cd <your local git repository>
$ git remote add upstream <https://github.com/<git-id>/<quickstart-repo>.git>
$ git push -u upstream master
```

3. Commit and push your source code to GitHub:

```
$ cd <your local repository>
$ git commit -am "added code for my app"
$ git push origin master
```

4. Create a new application in v3:

```
$ oc new-app --template=<template> \
-p SOURCE_REPOSITORY_URL=<https://github.com/<git-id>/<quickstart_repo>.git> \
-p DATABASE_USER=<your_db_user> \
-p DATABASE_NAME=<your_db_name> \
-p DATABASE_PASSWORD=<your_db_password> \
-p DATABASE_ADMIN_PASSWORD=<your_db_admin_password> 1
```

1

Only applicable for MongoDB.

You should now have 2 pods running, a web framework pod, and a database pod. The web framework pod environment should match the database pod environment. You can list the environment variables with `oc set env pod/<pod_name> --list`:

- ✎ **DATABASE_NAME** is now **<DB_SERVICE>_DATABASE**
- ✎ **DATABASE_USER** is now **<DB_SERVICE>_USER**
- ✎ **DATABASE_PASSWORD** is now **<DB_SERVICE>_PASSWORD**
- ✎ **DATABASE_ADMIN_PASSWORD** is now **MONGODB_ADMIN_PASSWORD** (only applicable for MongoDB)

If no **SOURCE_REPOSITORY_URL** is specified, the template will use the template URL (<https://github.com/openshift/<quickstart>-ex>) listed above as the source repository, and a **hello-welcome** application will be started.

5. If you are migrating a database, export databases to a dump file and restore the database in the new v3 database pod. Refer to the steps outlined in [Database Applications](#), skipping the `oc new-app` step as the database pod is already up and running.

6.5. CONTINUOUS INTEGRATION AND DEPLOYMENT (CI/CD)

6.5.1. Overview

This topic reviews the differences in continuous integration and deployment (CI/CD) applications between OpenShift version 2 (v2) and OpenShift version 3 (v3) and how to migrate these applications into the v3 environment.

6.5.2. Jenkins

The Jenkins applications in OpenShift version 2 (v2) and OpenShift version 3 (v3) are configured differently due to fundamental differences in architecture. For example, in v2, the application uses an integrated Git repository that is hosted in the gear to store the source code. In v3, the source code is located in a public or private Git repository that is hosted outside of the pod.

Furthermore, in OpenShift v3, Jenkins jobs can not only be triggered by source code changes, but also by changes in ImageStream, which are changes on the images that are used to build the application along with its source code. As a result, it is highly recommended that you migrate the Jenkins application manually by creating a new Jenkins application in v3, and then re-creating jobs with the configurations that are suitable to OpenShift v3 environment.

Consult these resources for more information on how to create a Jenkins application, configure jobs, and use Jenkins plug-ins properly:

- ✎ <https://github.com/openshift/origin/blob/master/examples/jenkins/README.md>
- ✎ <https://github.com/openshift/jenkins-plugin/blob/master/README.md>
- ✎ <https://github.com/openshift/origin/blob/master/examples/sample-app/README.md>

6.6. WEBHOOKS AND ACTION HOOKS

6.6.1. Overview

This topic reviews the differences in webhooks and action hooks between OpenShift version 2 (v2) and OpenShift version 3 (v3) and how to migrate these applications into the v3 environment.

6.6.2. Webhooks

1. After creating a **BuildConfig** from a GitHub repository, run:

```
$ oc describe bc/<name-of-your-BuildConfig>
```

This will output a webhook GitHub URL that looks like:

```
<https://api.dev-preview-  
int.openshift.com:443/oapi/v1/namespaces/nsname/buildconfigs/bcna  
me/webhooks/secret/github>.
```

2. Cut and paste this URL into GitHub, from the GitHub web console.
3. In your GitHub repository, select **Add Webhook** from **Settings** → **Webhooks & Services**.
4. Paste the URL output (similar to above) into the **Payload URL** field.

You should see a message from GitHub stating that your webhook was successfully configured.

Now, whenever you push a change to your GitHub repository, a new build will automatically start, and upon a successful build a new deployment will start.



Note

If you delete or recreate your application, you will have to update the **Payload URL** field in GitHub with the new **BuildConfig** webhook url.

6.6.3. Action Hooks

In OpenShift version 2 (v2), there are build, deploy, post_deploy, and pre_build scripts or action_hooks that are located in the **.openshift/action_hooks** directory. While there is no one-to-one mapping of function for these in v3, the [S2I tool](#) in v3 does have the option of adding [customizable scripts](#), either in a designated URL or in the **.s2i/bin** directory of your source repository.

OpenShift version 3 (v3) also offers a [post-build hook](#) for running basic testing of an image after it is built and before it is pushed to the registry. [Deployment hooks](#) are configured in the deployment configuration.

In v2, action_hooks are commonly used to set up environment variables. In v2, any environment variables should be passed with:

```
$ oc new-app <source-url> -e ENV_VAR=env_var
```

or:

```
$ oc new-app <template-name> -p ENV_VAR=env_var
```

Also, environment variables can be added or changed using:

```
$ oc set env dc/<name-of-dc>  
ENV_VAR1=env_var1 ENV_VAR2=env_var2'
```

6.7. S2I TOOL

6.7.1. Overview

The [Source-to-Image \(S2I\) tool](#) injects application source code into a container image and the final product is a new and ready-to-run container image that incorporates the builder image and built source code. The S2I tool can be installed on your local machine without OpenShift Enterprise from [the repository](#).

The S2I tool is a very powerful tool to test and verify your application and images locally before using them on OpenShift Enterprise.

6.7.2. Creating a Container Image

1. Identify the builder image that is needed for the application. Red Hat offers multiple builder images for different languages including [Python](#), [Ruby](#), [Perl](#), [PHP](#), and [Node.js](#). Other images are available from [the community space](#).
2. S2I can build images from source code in a local file system or from a Git repository. To build a new container image from the builder image and the source code:

```
$ s2i build <source-location> <builder-image-name> <output-image-name>
```



Note

<source-location> can either be a Git repository URL or a directory to source code in a local file system.

3. Test the built image with the Docker daemon:

```
$ docker run -d --name <new-name> -p <port-number>:<port-number>  
<output-image-name>  
$ curl localhost:<port-number>
```

4. Push the new image to the [OpenShift registry](#).
5. Create a new application from the image in the OpenShift registry using the **oc** command:

```
$ oc new-app <image-name>
```

6.8. SUPPORT GUIDE

6.8.1. Overview

This topic reviews supported languages, frameworks, databases, and markers for OpenShift version

2 (v2) and OpenShift version 3 (v3).

6.8.2. Supported Databases

See the [Supported Databases](#) section of the Database Applications topic.

6.8.3. Supported Languages

- ✳ [PHP](#)
- ✳ [Python](#)
- ✳ [Perl](#)
- ✳ [Node.js](#)
- ✳ [Ruby](#)
- ✳ [JBoss/xPaaS](#)

6.8.4. Supported Frameworks

Table 6.1. Supported Frameworks

v2	v3
Jenkins Server	jenkins-persistent
Drupal 7	
Ghost 0.7.5	
WordPress 4	
Ceylon	
Go	
MEAN	

6.8.5. Supported Markers

Table 6.2. Python

v2	v3
pip_install	If your repository contains requirements.txt , then pip is invoked by default. Otherwise, pip is not used.

Table 6.3. Ruby

v2	v3
disable_asset_compilation	This can be done by setting DISABLE_ASSET_COMPILATION environment variable to true on the buildconfig strategy definition.

Table 6.4. Perl

v2	v3
enable_cpan_tests	This can be done by setting ENABLE_CPAN_TEST environment variable to true on the build configuration .

Table 6.5. PHP

v2	v3
use_composer	composer is always used if the source repository includes a composer.json in the root directory.

Table 6.6. Node.js

v2	v3
NODEJS_VERSION	N/A

v2	v3
use_npm	npm is always used to start the application, unless DEV_MODE is set to true , in which case nodemon is used instead.

Table 6.7. JBoss EAP, JBoss WS, WildFly

v2	v3
enable_debugging	This option is controlled via the ENABLE_JPDA environment variable set on the deployment configuration by setting it to any non-empty value.
skip_maven_build	If <i>pom.xml</i> is present, maven will be run.
java7	N/A
java8	JavaEE is using JDK8.

Table 6.8. Jenkins

v2	v3
enable_debugging	N/A

Table 6.9. All

v2	v3
force_clean_build	There is a similar concept in v3, as noCache field in buildconfig forces the container build to rerun each layer. In the S2I build, the incremental flag is false by default, which indicates a clean build .
hot_deploy	Ruby , Python , Perl , PHP , Node.js

v2	v3
enable_public_server_status	N/A
disable_auto_scaling	Autoscaling is off by default and it can be turn on via pod auto-scaling .

6.8.6. Supported Environment Variables

✎ [MySQL](#)

✎ [MongoDB](#)

✎ [PostgreSQL](#)

CHAPTER 7. APPLICATION TUTORIALS

7.1. OVERVIEW

This topic group includes information on how to get your application up and running in OpenShift Enterprise and covers different languages and their frameworks.

7.2. QUICKSTART TEMPLATES

7.2.1. Overview

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

To explore a quickstart, create an application from a template. Your administrator may have already installed these templates in your OpenShift Enterprise cluster, in which case you can simply select it from the web console. See the [template](#) documentation for more information on how to upload, create from, and modify a template.

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.

7.2.2. Web Framework Quickstart Templates

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

- ✎ CakePHP: a PHP web framework (includes a MySQL database)
 - [Template definition](#)
 - [Source repository](#)
- ✎ Dancer: a Perl web framework (includes a MySQL database)
 - [Template definition](#)
 - [Source repository](#)
- ✎ Django: a Python web framework (includes a PostgreSQL database)
 - [Template definition](#)
 - [Source repository](#)
- ✎ NodeJS: a NodeJS web application (includes a MongoDB database)
 - [Template definition](#)
 - [Source repository](#)

- ✳️ Rails: a Ruby web framework (includes a PostgreSQL database)

- [Template definition](#)
- [Source repository](#)

7.3. RUBY ON RAILS

7.3.1. Overview

Ruby on Rails is a popular web framework written in [Ruby](#). This guide covers using Rails 4 on OpenShift Enterprise.

Warning

We strongly advise going through the whole tutorial to have an overview of all the steps necessary to run your application on the OpenShift Enterprise. 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.

For this guide you will need:

- ✳️ Basic Ruby/Rails knowledge
- ✳️ Locally installed version of Ruby 2.0.0+, Rubygems, Bundler
- ✳️ Basic Git knowledge
- ✳️ Running instance of OpenShift Enterprise v3

7.3.2. Local Workstation Setup

First make sure that an instance of OpenShift Enterprise is running and is available. For more info on how to get OpenShift Enterprise up and running check the [installation methods](#). Also make sure that your [oc CLI client is installed](#) and the command is accessible from your command shell, so you can use it to [log in](#) using your email address and password.

7.3.2.1. Setting Up the Database

Rails applications are almost always used with a database. For the local development we chose the PostgreSQL database. To install it type:

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

Next you need to initialize the database with:

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

This command will create the `/var/lib/pgsql/data` directory, in which the data will be stored.

Start the database by typing:

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

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

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

Note that the user we created has no password.

7.3.3. Writing Your Application

If you are starting your Rails application from scratch, you need to install the Rails gem first.

```
$ gem install rails
Successfully installed rails-4.2.0
1 gem installed
```

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

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

Then change into your new application directory.

```
$ cd rails-app
```

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'
```

To generate a new **Gemfile.lock** with all your dependencies run:

```
$ bundle install
```

In addition to using the **postgresql** database with the **pg** gem, you'll also need to ensure 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:
```

Create your application's development and test databases by using this **rake** command:

```
$ rake db:create
```

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

7.3.3.1. Creating a Welcome Page

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

In order to have a custom welcome page we need to 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 this necessary steps for you.

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

All the necessary files have been created, now we just need to edit line 2 in **config/routes.rb** file to look like:

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

Run the rails server to verify the page is available.

```
$ rails server
```

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

7.3.3.2. Configuring the Application for OpenShift Enterprise

In order to have your application communicating with the PostgreSQL database service that will be running in OpenShift Enterprise, you will need to edit the **default** section in your **config/database.yml** to use [environment variables](#), which you will define later, upon the database service creation.

The **default** section in your edited **config/database.yml** together with pre-defined variables should look like:

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

For an example of how the final file should look, see [Ruby on Rails example application config/database.yml](#).

7.3.3.3. Storing Your Application in Git

OpenShift Enterprise requires [git](#), if you don't have it installed you will need to install it.

Building an application in OpenShift Enterprise usually requires that the source code be stored in a [git](#) repository, so you will need to install **git** if you do not already have it.

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

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

Now run these commands in your Rails app directory to initialize and commit your code to git:

```
$ git init
$ git add .
$ git commit -m "initial commit"
```

Once your application is committed you need to push it to a remote repository. For this you would need a [GitHub account](#), in which you [create a new repository](#).

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

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

After that, push your application to your remote git repository.

```
$ git push
```

7.3.4. Deploying Your Application to OpenShift Enterprise

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"
```

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

Deploying your application in OpenShift Enterprise involves three steps:

- ✎ Creating a database [service](#) from OpenShift Enterprise's [PostgreSQL image](#)
- ✎ Creating a frontend [service](#) from OpenShift Enterprise's [Ruby 2.0 builder image](#) and your Ruby on Rails source code, which we wire with the database service
- ✎ Creating a route for your application.

7.3.4.1. Creating the Database Service

Your Rails application expects a running database [service](#). For this service use [PostgreSQL database image](#).

To create the database [service](#) you will use the `oc new-app` command. To this command you will need to 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 we are going to be setting 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

For example:

```
$ 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
```

To watch the progress of this command:

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

7.3.4.2. Creating the Frontend Service

To bring your application to OpenShift Enterprise, you need to specify a repository in which your application lives, using once again the `oc new-app` command, in which you will need to specify database related [environment variables](#) we setup in the [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
```

With this command, OpenShift Enterprise fetches the source code, sets up the Builder image, [builds](#) your application image, and deploys the newly created image together with the specified [environment variables](#). The application is named **rails-app**.

You can 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"
  }
],
```

To check the [build](#) process, use the [build-logs](#) command:

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

Once the [build](#) is complete, you can look at the running [pods](#) in OpenShift Enterprise:

```
$ oc get pods
```

You should see a line starting with myapp-(#number)-(some hash) and that is your application running in OpenShift Enterprise.

Before your application will be functional, you need to initialize the database by running the database migration script. There are two ways you can do this:

- ✱ Manually from the running frontend container:

First you need to exec into frontend container with [rsh](#) command:

```
$ oc rsh <FRONTEND_POD_ID>
```

Run the migration from inside the container:

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

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

- ✎ By adding pre-deployment [lifecycle hooks](#) in your template. For example check the [hooks example](#) in our [Rails example](#) application.

7.3.4.3. Creating a Route for Your Application

To expose a service by giving it an externally-reachable hostname like **www.example.com** use OpenShift Enterprise [route](#). In your case you need to expose the frontend service by typing:

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

Warning

It's the user's responsibility to ensure the hostname they specify resolves into the IP address of the router. For more information, check the OpenShift Enterprise documentation on:

✎ [Routes](#)

✎ [Configuring a Highly-available Routing Service](#)

CHAPTER 8. OPENING A REMOTE SHELL TO CONTAINERS

8.1. OVERVIEW

The **oc rsh** command allows you to locally access and manage tools that are on the system. The secure shell (SSH) is the underlying technology and industry standard that provides a secure connection to the application. Access to applications with the shell environment is protected and restricted with Security-Enhanced Linux (SELinux) policies.

8.2. START A SECURE SHELL SESSION

Open a remote shell session to a container:

```
$ oc rsh <pod>
```

While in the remote shell, you can issue commands as if you are inside the container and perform local operations like monitoring, debugging, and using CLI commands specific to what is running in the container.

For example, in a MySQL container, you can count the number of records in the database by invoking the **mysql** command, then using the prompt to type in the **SELECT** command. You can also use commands like **ps(1)** and **ls(1)** for validation.

BuildConfigs and **DeployConfigs** map out how you want things to look and pods (with containers inside) are created and dismantled as needed. Your changes are not persistent. If you make changes directly within the container and that container is destroyed and rebuilt, your changes will no longer exist.



Note

oc exec can be used to execute a command remotely. However, the **oc rsh** command provides an easier way to keep a remote shell open persistently.

8.3. SECURE SHELL SESSION HELP

For help with usage, options, and to see examples:

```
$ oc rsh -h
```

CHAPTER 9. TEMPLATES

9.1. OVERVIEW

A [template](#) describes a set of [objects](#) that can be parameterized and processed to produce a list of objects for creation by OpenShift Enterprise. A template can be processed to create anything you have permission to create within a project, for example [services](#), [build configurations](#), and [deployment configurations](#). A template 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](#).

9.2. 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.

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

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

You can upload a template to a different project using the **-n** option with the name of the project:

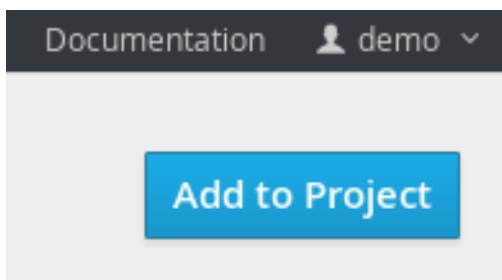
```
$ oc create -f <filename> -n <project>
```

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

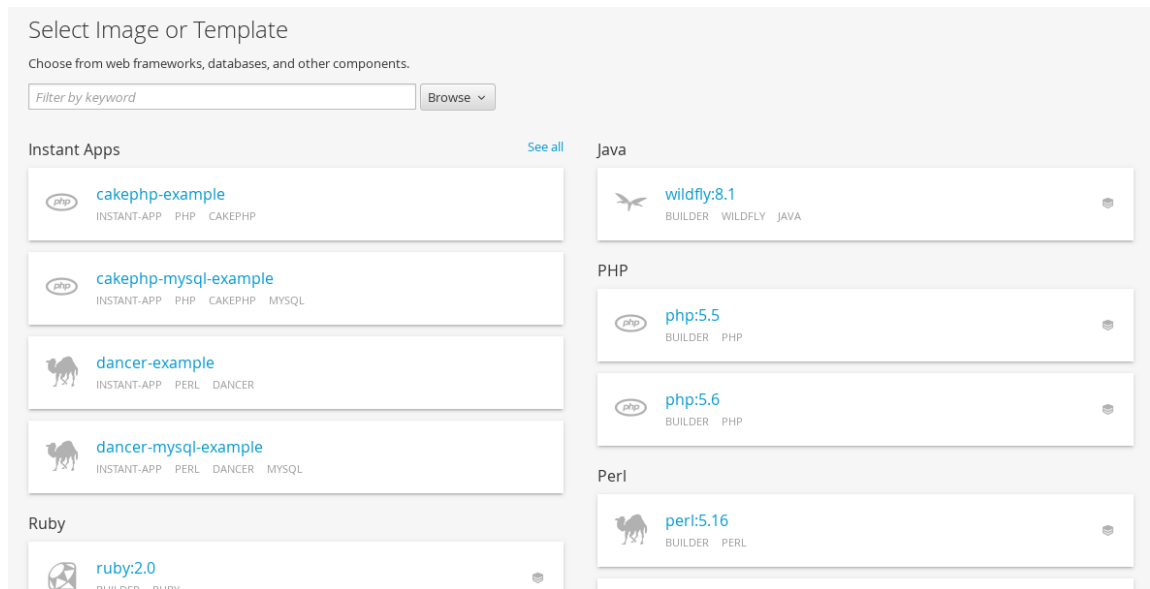
9.3. CREATING FROM TEMPLATES USING THE WEB CONSOLE

To create the objects from an [uploaded template](#) using the web console:


1. While in the desired project, click **Add to Project**



2. Select a template from the list of templates in your project, or provided by the global template library:



3. Modify template parameters in the template creation screen:



rails-postgresql-example 1

Namespace: openshift
An example Rails application with a PostgreSQL database

Images 2

openshift/ruby:2.0

rails-example

openshift/postgresql-92-centos7

Parameters 3

Edit Parameters

SOURCE_REPOSITORY_URL

https://github.com/openshift/rails-ex.git

The URL of the repository with your application source code

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

rails-example.openshiftapps.com

The exposed hostname that will route to the Rails service

GITHUB_WEBHOOK_SECRET

(generated)

A secret string used to configure the GitHub webhook

SECRET_KEY_BASE

(generated)

Your secret key for verifying the integrity of signed cookies

APPLICATION_USER

openshift

The application user that is used within the sample application to authorize access on pages

APPLICATION_PASSWORD

secret

The application password that is used within the sample application to authorize access on pages

DATABASE_SERVICE_NAME

postgresql

Database service name

POSTGRESQL_USER

(generated)

database username

POSTGRESQL_PASSWORD

(generated)

database password

POSTGRESQL_DATABASE

root

database name

POSTGRESQL_MAX_CONNECTIONS

10

database max connections

POSTGRESQL_SHARED_BUFFERS

12MB

database shared buffers

Labels 4

Edit labels About labels

Each label is applied to each created resource.

template

rails-postgresql-example

Create

Cancel

Template name and description.

Container images included in the template.

Parameters defined by the template. You can edit values for parameters defined in the template here.

Labels to assign to all items included in the template. You can add and edit labels

62

for objects.

9.4. CREATING FROM TEMPLATES USING THE CLI

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

9.4.1. 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.

There is also the ability to add labels in the template from the command line.

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

9.4.2. Parameters

The list of parameters that you can override are listed in the [parameters section of the template](#). You can list them 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/openshift/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.

9.4.3. 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:

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

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

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

The **process** command also takes a list of templates you can process to a list of objects. In that case, every template will be processed and the resulting list of objects will contain objects from all templates passed to a process command:

```
$ cat <first_template> <second_template> | oc process -f -
```

You can 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 -
```

You can override any [parameter](#) values defined in the file by adding the **-v** option followed by a comma-separated list of **<name>=<value>** pairs. 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:

Example 9.1. Creating a List of Objects from a Template

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

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 \
-v POSTGRESQL_USER=bob,POSTGRESQL_DATABASE=mydatabase \
| oc create -f -
```

9.5. MODIFYING AN UPLOADED TEMPLATE

You can edit a template that has already been uploaded to your project by using the following command:

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

9.6. USING THE INSTANT APP AND QUICKSTART TEMPLATES

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

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

If they are not available, direct your cluster administrator to the [Loading the Default Image Streams and Templates](#) topic.

By default, the templates build using a public source repository on [GitHub](#) that contains the necessary application code. In order to be able to modify the source and build your own version of the application, you must:

1. Fork the repository referenced by the template's default **SOURCE_REPOSITORY_URL** parameter.
2. 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. A walkthrough of this process using the web console is provided in [Getting Started for Developers: Web Console](#).



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.

9.7. 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.

9.7.1. Description

The template description covers information that informs users what your template does and helps them find it when searching in the web console. In addition to general descriptive information, it includes a set of tags. Useful tags include the name of the language your template is related to (e.g., **java**, **php**, **ruby**, etc.). In addition, adding the special tag **instant-app** causes your template to be displayed in the list of Instant Apps on the template selection page of the web console.

```
kind: "Template"
apiVersion: "v1"
metadata:
  name: "cakephp-mysql-example" 1
  annotations:
    description: "An example CakePHP application with a MySQL database" 2
  tags: "instant-app,php,cakephp,mysql" 3
  iconClass: "icon-php" 4
```

1

The name of the template as it will appear to users.

2

A description of the template.

3

Tags to be associated with the template for searching and grouping.

4

An icon to be displayed with your template in the web console.

9.7.2. 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.

```
kind: "Template"
apiVersion: "v1"
...
labels:
  template: "cakephp-mysql-example" 1
```

1

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

9.7.3. Parameters

Parameters allow a value to be supplied by the user or generated when the template is instantiated. 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 by placing values in the form "**`${PARAMETER_NAME}`**" in place of any string field in the template.

```
kind: Template
apiVersion: v1
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}"
      parameters:
        - name: SOURCE_REPOSITORY_URL 2
          description: The URL of the repository with your application source
            code 3
          value: https://github.com/openshift/cakephp-ex.git 4
          required: true 5
        - name: GITHUB_WEBHOOK_SECRET
          description: A secret string used to configure the GitHub webhook
          generate: expression 6
          from: "[a-zA-Z0-9]{40}" 7
```

1

This value will be replaced with the value of the **SOURCE_REPOSITORY_URL** parameter when the template is instantiated.

2

The name of the parameter. This value is displayed to users and used to reference the parameter within the template.

3

A description of the parameter.

4

A default value for the parameter which will be used if the user does not override the value when instantiating the template.

5

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.

6

A parameter which has its value generated via a [regular expression-like syntax](#).

7

The input to the generator. In this case, the generator will produce a 40 character alphanumeric value including upper and lowercase characters.

9.7.4. 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.

```
kind: "Template"
apiVersion: "v1"
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 **namespace** field, the field will be stripped out of the definition during template instantiation. This is necessary because all objects created during instantiation are placed into the target namespace, so it would be invalid for the object to declare a different namespace.

9.7.5. Creating a Template from Existing Objects

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

```
$ oc export all --as-template=<template_name> > <template_filename>
```

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

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

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

CHAPTER 10. SERVICE ACCOUNTS

10.1. OVERVIEW

When a person uses the OpenShift Enterprise CLI or web console, their API token authenticates them to the OpenShift API. However, when a regular user's credentials are not available, it is common for components to make API calls independently. For example:

- ✧ Replication controllers make API calls to create or delete pods.
- ✧ Applications inside containers could make API calls for discovery purposes.
- ✧ External applications could make API calls for monitoring or integration purposes.

Service accounts provide a flexible way to control API access without sharing a regular user's credentials.

10.2. USER NAMES AND GROUPS

Every service account has an associated user name that can be granted roles, just like a regular user. The user name is derived from its project and name:

```
system:serviceaccount:<project>:<name>
```

For example, to add the **view** role to the **robot** service account in the **top-secret** project:

```
$ oc policy add-role-to-user view system:serviceaccount:top-secret:robot
```

Every service account is also a member of two groups:

system:serviceaccounts

Includes all service accounts in the system.

system:serviceaccounts:<project>

Includes all service accounts in the specified project.

For example, to allow all service accounts in all projects to view resources in the **top-secret** project:

```
$ oc policy add-role-to-group view system:serviceaccounts -n top-secret
```

To allow all service accounts in the **managers** project to edit resources in the **top-secret** project:

```
$ oc policy add-role-to-group edit system:serviceaccounts:managers -n top-secret
```

10.3. DEFAULT SERVICE ACCOUNTS AND ROLES

Three service accounts are automatically created in every project:

Service Account	Usage
builder	Used by build pods. It is given the system:image-builder role, which allows pushing images to any image stream in the project using the internal Docker registry.
deployer	Used by deployment pods and is given the system:deployer role, which allows viewing and modifying replication controllers and pods in the project.
default	Used to run all other pods unless they specify a different service account.

All service accounts in a project are given the **system:image-puller** role, which allows pulling images from any image stream in the project using the internal Docker registry.

10.4. MANAGING SERVICE ACCOUNTS

Service accounts are API objects that exist within each project. They can be created or deleted like any other API object.

```
$ oc create serviceaccount robot
serviceaccounts/robot
```

10.5. MANAGING SERVICE ACCOUNT CREDENTIALS

As soon as a service account is created, two secrets are automatically added to it:

- ✎ an API token
- ✎ credentials for the internal Docker registry

These can be seen by describing the service account:

```
$ oc describe serviceaccount robot
Name:                robot
Labels:              <none>
Image pull secrets:  robot-dockercfg-624cx

Mountable secrets:   robot-token-uzkbh
                    robot-dockercfg-624cx

Tokens:              robot-token-8bhpp
                    robot-token-uzkbh
```

The system ensures that service accounts always have an API token and internal Docker registry credentials.

The generated API token and Docker registry credentials do not expire, but they can be revoked by deleting the secret. When the secret is deleted, a new one is automatically generated to take its place.

10.6. MANAGING ALLOWED SECRETS

In addition to providing API credentials, a pod's service account determines which secrets the pod is allowed to use.

Pods use secrets in two ways:

- ✧ image pull secrets, providing credentials used to pull images for the pod's containers
- ✧ mountable secrets, injecting the contents of secrets into containers as files

To allow a secret to be used as an image pull secret by a service account's pods, run:

```
$ oc secrets add --for=pull \
  serviceaccount/<serviceaccount-name> \
  secret/<secret-name>
```

To allow a secret to be mounted by a service account's pods, run:

```
$ oc secrets add --for=mount \
  serviceaccount/<serviceaccount-name> \
  secret/<secret-name>
```

This example creates and adds secrets to a service account:

```
$ oc secrets new secret-plans plan1.txt plan2.txt
secret/secret-plans

$ oc secrets new-dockercfg my-pull-secret \
  --docker-username=mastermind \
  --docker-password=12345 \
  --docker-email=mastermind@example.com
secret/my-pull-secret

$ oc secrets add serviceaccount/robot secret/secret-plans --for=mount

$ oc secrets add serviceaccount/robot secret/my-pull-secret --for=pull

$ oc describe serviceaccount robot
Name:          robot
Labels:        <none>
Image pull secrets: robot-dockercfg-624cx
                  my-pull-secret

Mountable secrets: robot-token-uzkbh
                  robot-dockercfg-624cx
                  secret-plans

Tokens:        robot-token-8bhpp
                  robot-token-uzkbh
```

10.7. USING A SERVICE ACCOUNT'S CREDENTIALS INSIDE A CONTAINER

When a pod is created, it specifies a service account (or uses the default service account), and is allowed to use that service account's API credentials and referenced secrets.

A file containing an API token for a pod's service account is automatically mounted at ***/var/run/secrets/kubernetes.io/serviceaccount/token***.

That token can be used to make API calls as the pod's service account. This example calls the ***users/~*** API to get information about the user identified by the token:

```
$ TOKEN="$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)"

$ curl --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt \
    "https://openshift.default.svc.cluster.local/oapi/v1/users/~" \
    -H "Authorization: Bearer $TOKEN"

kind: "User"
apiVersion: "v1"
metadata:
  name: "system:serviceaccount:top-secret:robot"
  selflink: "/oapi/v1/users/system:serviceaccount:top-secret:robot"
  creationTimestamp: null
identities: null
groups:
  - "system:serviceaccounts"
  - "system:serviceaccounts:top-secret"
```

10.8. USING A SERVICE ACCOUNT'S CREDENTIALS EXTERNALLY

The same token can be distributed to external applications that need to authenticate to the API.

Use the following syntax to view a service account's API token:

```
$ oc describe secret <secret-name>
```

For example:

```
$ oc describe secret robot-token-uzkbh -n top-secret
Name:  robot-token-uzkbh
Labels:  <none>
Annotations:  kubernetes.io/service-account.name=robot,kubernetes.io/service-account.uid=49f19e2e-16c6-11e5-afdc-3c970e4b7ffe

Type: kubernetes.io/service-account-token

Data

token: eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...
```

```
$ oc login --token=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...
```

Logged into "https://server:8443" as "system:serviceaccount:top-secret:robot" using the token provided.

You don't have any projects. You can try to create a new project, by running

```
$ oc new-project <projectname>
```

```
$ oc whoami  
system:serviceaccount:top-secret:robot
```

CHAPTER 11. BUILDS

11.1. OVERVIEW

A [build](#) is the process of transforming input parameters into a resulting object. Most often, the process is used to transform source code into a runnable image.

Build configurations are characterized by a strategy and one or more sources. The strategy determines the aforementioned process, while the sources provide its input.

There are three build strategies:

- ✳ Source-To-Image (S2I) ([description](#), [options](#))
- ✳ Docker ([description](#), [options](#))
- ✳ Custom ([description](#), [options](#))

And there are four types of build source:

- ✳ [Git](#)
- ✳ [Dockerfile](#)
- ✳ [Image](#)
- ✳ [Binary](#)

It is up to each build strategy to consider or ignore a certain type of source, as well as to determine how it is to be used.

Binary and Git are mutually exclusive source types. Dockerfile and Image can be used by themselves, with each other, or together with either Git or Binary. Also, the Binary build source type is unique from the other options in [how it is specified to the system](#).

11.2. DEFINING A BUILDCONFIG

A build configuration describes a single build definition and a set of [triggers](#) for when a new build should be created.

A build configuration is defined by a **BuildConfig**, which is a REST object that can be used in a POST to the API server to create a new instance. The following example **BuildConfig** results in a new build every time a container image tag or the source code changes:

Example 11.1. BuildConfig Object Definition

```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "ruby-sample-build" 1
spec:
  triggers: 2
  - type: "GitHub"
    github:
```

```

      secret: "secret101"
    - type: "Generic"
      generic:
        secret: "secret101"
    - type: "ImageChange"
source: 3
  type: "Git"
  git:
    uri: "https://github.com/openshift/ruby-hello-world"
    dockerfile: "FROM openshift/ruby-22-centos7\nUSER example"
strategy: 4
  type: "Source"
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "ruby-20-centos7:latest"
output: 5
  to:
    kind: "ImageStreamTag"
    name: "origin-ruby-sample:latest"
postCommit: 6
  script: "bundle exec rake test"

```

1

This specification will create a new **BuildConfig** named **ruby-sample-build**.

2

You can specify a list of [triggers](#), which cause a new build to be created.

3

The **source** section defines the source of the build. The **type** determines the primary source of input, and can be either **Git**, to point to a code repository location; **Dockerfile**, to build from an inline Dockerfile; or **Binary**, to accept binary payloads. Using multiple sources at once is possible. Refer to the documentation for each source type for details.

4

The **strategy** section describes the build strategy used to execute the build. You can specify **Source**, **Docker** and **Custom** strategies here. This above example uses the **ruby-20-centos7** container image that Source-To-Image will use for the application build.

5

After the container image is successfully built, it will be pushed into the repository.

After the container image is successfully built, it will be pushed into the repository described in the **output** section.

6

The **postCommit** section defines an optional [build hook](#).

11.3. SOURCE-TO-IMAGE STRATEGY OPTIONS

The following options are specific to the [S2I build strategy](#).

11.3.1. Force Pull

By default, if the builder image specified in the build configuration is available locally on the node, that image will be used. However, to override the local image and refresh it from the registry to which the image stream points, create a **BuildConfig** with the **forcePull** flag set to **true**:

```
strategy:
  type: "Source"
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "builder-image:latest" 1
    forcePull: true 2
```

1

The builder image being used, where the local version on the node may not be up to date with the version in the registry to which the image stream points.

2

This flag causes the local builder image to be ignored and a fresh version to be pulled from the registry to which the image stream points. Setting **forcePull** to **false** results in the default behavior of honoring the image stored locally.

11.3.2. Incremental Builds

S2I can perform incremental builds, which means it reuses artifacts from previously-built images. To create an incremental build, create a **BuildConfig** with the following modification to the strategy definition:

```
strategy:
  type: "Source"
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
```

```
name: "incremental-image:latest" 1
incremental: true 2
```

1

Specify an image that supports incremental builds. Consult the documentation of the builder image to determine if it supports this behavior.

2

This flag controls whether an incremental build is attempted. If the builder image does not support incremental builds, the build will still succeed, but you will get a log message stating the incremental build was not successful because of a missing **save-artifacts** script.



Note

See the [S2I Requirements](#) topic for information on how to create a builder image supporting incremental builds.

11.3.3. Overriding Builder Image Scripts

You can override the **assemble**, **run**, and **save-artifacts** [S2I scripts](#) provided by the builder image in one of two ways. Either:

1. Provide an **assemble**, **run**, and/or **save-artifacts** script in the **.s2i/bin** directory of your application source repository, or
2. Provide a URL of a directory containing the scripts as part of the strategy definition. For example:

```
strategy:
  type: "Source"
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "builder-image:latest"
    scripts: "http://somehost.com/scripts_directory" 1
```

1

This path will have **run**, **assemble**, and **save-artifacts** appended to it. If any or all scripts are found they will be used in place of the same named script(s) provided in the image.



Note

Files located at the **scripts** URL take precedence over files located in **.s2i/bin** of the source repository. See the [S2I Requirements](#) topic and the [S2I documentation](#) for information on how S2I scripts are used.

11.3.4. Environment Variables

There are two ways to make environment variables available to the [source build](#) process and resulting image: [environment files](#) and [BuildConfig environment](#) values.

11.3.4.1. Environment Files

Source build enables you to set environment values (one per line) inside your application, by specifying them in a **.s2i/environment** file in the source repository. The environment variables specified in this file are present during the build process and in the final container image. The complete list of supported environment variables is available in the [documentation](#) for each image.

If you provide a **.s2i/environment** file in your source repository, S2I reads this file during the build. This allows customization of the build behavior as the **assemble** script may use these variables.

For example, if you want to disable assets compilation for your Rails application, you can add **DISABLE_ASSET_COMPILATION=true** in the **.s2i/environment** file to cause assets compilation to be skipped during the build.

In addition to builds, the specified environment variables are also available in the running application itself. For example, you can add **RAILS_ENV=development** to the **.s2i/environment** file to cause the Rails application to start in **development** mode instead of **production**.

11.3.4.2. BuildConfig Environment

You can add environment variables to the **sourceStrategy** definition of the **BuildConfig**. The environment variables defined there are visible during the **assemble** script execution and will be defined in the output image, making them also available to the **run** script and application code.

For example disabling assets compilation for your Rails application:

```
sourceStrategy:
  ...
  env:
    - name: "DISABLE_ASSET_COMPILATION"
      value: "true"
```

11.4. DOCKER STRATEGY OPTIONS

The following options are specific to the [Docker build strategy](#).

11.4.1. FROM Image

The **FROM** instruction of the **Dockerfile** will be replaced by the **from** of the **BuildConfig**:

```
strategy:
  type: Docker
  dockerStrategy:
    from:
      kind: "ImageStreamTag"
      name: "debian:latest"
```

11.4.2. Dockerfile Path

By default, Docker builds use a Dockerfile (named **Dockerfile**) located at the root of the context specified in the **BuildConfig.spec.source.contextDir** field.

The **dockerfilePath** field allows the build to use a different path to locate your Dockerfile, relative to the **BuildConfig.spec.source.contextDir** field. It can be simply a different file name other than the default **Dockerfile** (for example, **MyDockerfile**), or a path to a Dockerfile in a subdirectory (for example, **dockerfiles/app1/Dockerfile**):

```
strategy:
  type: Docker
  dockerStrategy:
    dockerfilePath: dockerfiles/app1/Dockerfile
```

11.4.3. No Cache

Docker builds normally reuse cached layers found on the host performing the build. Setting the **noCache** option to **true** forces the build to ignore cached layers and rerun all steps of the **Dockerfile**:

```
strategy:
  type: "Docker"
  dockerStrategy:
    noCache: true
```

11.4.4. Force Pull

By default, if the builder image specified in the build configuration is available locally on the node, that image will be used. However, to override the local image and refresh it from the registry to which the image stream points, create a **BuildConfig** with the **forcePull** flag set to **true**:

```
strategy:
  type: "Docker"
  dockerStrategy:
    forcePull: true 1
```

This flag causes the local builder image to be ignored, and a fresh version to be pulled from the registry to which the image stream points. Setting **forcePull** to **false** results in the default behavior of honoring the image stored locally.

11.4.5. Environment Variables

To make environment variables available to the [Docker build](#) process and resulting image, you can add environment variables to the **dockerStrategy** definition of the **BuildConfig**.

The environment variables defined there are inserted as a single **ENV** Dockerfile instruction right after the **FROM** instruction, so that it can be referenced later on within the Dockerfile.

The variables are defined during build and stay in the output image, therefore they will be present in any container that runs that image as well.

For example, defining a custom HTTP proxy to be used during build and runtime:

```
dockerStrategy:
  ...
  env:
    - name: "HTTP_PROXY"
      value: "http://myproxy.net:5187/"
```

Cluster administrators can also [configure global build settings using Ansible](#).

11.5. CUSTOM STRATEGY OPTIONS

The following options are specific to the [Custom build strategy](#).

11.5.1. FROM Image

Use the **customStrategy.from** section to indicate the image to use for the custom build:

```
strategy:
  type: "Custom"
  customStrategy:
    from:
      kind: "DockerImage"
      name: "openshift/sti-image-builder"
```

11.5.2. Exposing the Docker Socket

In order to allow the running of Docker commands and the building of container images from inside the container, the build container must be bound to an accessible socket. To do so, set the **exposeDockerSocket** option to **true**:

```
strategy:
  type: "Custom"
  customStrategy:
    exposeDockerSocket: true
```

11.5.3. Secrets

In addition to [secrets](#) for [source](#) and [images](#) that can be added to all build types, custom strategies allow adding an arbitrary list of secrets to the builder pod.

Each secret can be mounted at a specific location:

```
strategy:
  type: "Custom"
  customStrategy:
    secrets:
      - secretSource: 1
        name: "secret1"
        mountPath: "/tmp/secret1" 2
      - secretSource:
        name: "secret2"
        mountPath: "/tmp/secret2"
```

1

secretSource is a reference to a secret in the same namespace as the build.

2

mountPath is the path inside the custom builder where the secret should be mounted.

11.5.4. Force Pull

By default, when setting up the build pod, the build controller checks if the image specified in the build configuration is available locally on the node. If so, that image will be used. However, to override the local image and refresh it from the registry to which the image stream points, create a **BuildConfig** with the **forcePull** flag set to **true**:

```
strategy:
  type: "Custom"
  customStrategy:
    forcePull: true 1
```

1

This flag causes the local builder image to be ignored, and a fresh version to be pulled from the registry to which the image stream points. Setting **forcePull** to **false** results in the default behavior of honoring the image stored locally.

11.5.5. Environment Variables

To make environment variables available to the [Custom build](#) process, you can add environment variables to the **customStrategy** definition of the **BuildConfig**.

The environment variables defined there are passed to the pod that runs the custom build.

For example, defining a custom HTTP proxy to be used during build:

```
customStrategy:
  ...
  env:
    - name: "HTTP_PROXY"
      value: "http://myproxy.net:5187/"
```

Cluster administrators can also [configure global build settings using Ansible](#).

11.6. BUILD INPUTS

There are several ways to provide content for builds to operate on. In order of precedence:

- ✧ Inline Dockerfile definitions
- ✧ Content extracted from existing images
- ✧ Git repositories
- ✧ Binary inputs

These can be combined into a single build. As the inline Dockerfile takes precedence, it can overwrite any other file named Dockerfile provided by another input. Binary input and Git repository are mutually exclusive inputs.

When the build is run, a working directory is constructed and all input content is placed in the working directory (e.g. the input git repository is cloned into the working directory, files specified from input images are copied into the working directory using the target path). Next the build process will **cd** into the **contextDir** if one is defined. Then the inline **Dockerfile** (if any) is written to the current directory. Last, the content from the current directory will be provided to the build process for reference by the **Dockerfile**, **assemble** script, or custom builder logic. This means any input content that resides outside the **contextDir** will be ignored by the build.

Here is an example of a source definition that includes multiple input types and an explanation of how they are combined. For more details on how each input type is defined, see the specific sections for each input type.

```
source:
  git:
    uri: https://github.com/openshift/ruby-hello-world.git 1
  images:
    - from:
        kind: ImageStreamTag
        name: myinputimage:latest
        namespace: mynamespace
    paths:
```

```

- destinationDir: app/dir/injected/dir 2
  sourcePath: /usr/lib/somefile.jar
contextDir: "app/dir" 3
dockerfile: "FROM centos:7\nRUN yum install -y httpd" 4

```

1

The repository to be cloned into the working directory for the build

2

`/usr/lib/somefile.jar` from `myinputimage` will be stored in
`<workingdir>/app/dir/injected/dir`

3

The working dir for the build will become `<original_workingdir>/app/dir`

4

A **Dockerfile** with this content will be created in `<original_workingdir>/app/dir`,
 overwriting any existing file with that name

11.7. GIT REPOSITORY SOURCE OPTIONS

When the **BuildConfig.spec.source.type** is **Git**, a Git repository is required, and an inline Dockerfile is optional.

The source code is fetched from the location specified and, if the **BuildConfig.spec.source.dockerfile** field is specified, the inline Dockerfile replaces the one in the **contextDir** of the Git repository.

The source definition is part of the **spec** section in the **BuildConfig**:

```

source:
  type: "Git"
  git: 1
    uri: "https://github.com/openshift/ruby-hello-world"
    ref: "master"
  contextDir: "app/dir" 2
  dockerfile: "FROM openshift/ruby-22-centos7\nUSER example" 3

```

1

The **git** field contains the URI to the remote Git repository of the source code. Optionally,
 specify the **ref** field to check out a specific Git reference. A valid **ref** can be a SHA1 tag or

specify the **ref** field to check out a specific Git reference. A valid **ref** can be a SHA1 tag or a branch name.

2

The **contextDir** field allows you to override the default location inside the source code repository where the build looks for the application source code. If your application exists inside a sub-directory, you can override the default location (the root folder) using this field.

3

If the optional **dockerfile** field is provided, it should be a string containing a Dockerfile that overwrites any Dockerfile that may exist in the source repository.

When using the Git repository as a source without specifying the **ref** field, OpenShift Enterprise performs a shallow clone (**--depth=1** clone). That means only the **HEAD** (usually the **master** branch) is downloaded. This results in repositories downloading faster, including the commit history.

A shallow clone is also used when the **ref** field is specified and set to an existing remote branch name. However, if you specify the **ref** field to a specific commit, the system will fallback to a regular Git clone operation and checkout the commit, because using the **--depth=1** option only works with named branch refs.

To perform a full Git clone of the **master** for the specified repository, set the **ref** to **master**.

11.7.1. Using a Proxy for Git Cloning

If your Git repository can only be accessed using a proxy, you can define the proxy to use in the **source** section of the **BuildConfig**. You can configure both a HTTP and HTTPS proxy to use. Both fields are optional.



Note

Your source URI must use the HTTP or HTTPS protocol for this to work.

```
source:
  type: Git
  git:
    uri: "https://github.com/openshift/ruby-hello-world"
    httpProxy: http://proxy.example.com
    httpsProxy: https://proxy.example.com
```

Cluster administrators can also [configure a global proxy for Git cloning using Ansible](#).

11.7.2. Using Private Repositories for Builds

Supply valid credentials to build an application from a private repository.

Currently two types of authentication are supported: basic username-password and SSH key based authentication.

11.7.2.1. Basic Authentication

Basic authentication requires either a combination of **username** and **password**, or a **token** to authenticate against the SCM server. A **CA certificate** file, or a **.gitconfig** file can be attached.

A **secret** is used to store your keys.

1. Create the **secret** first before using the username and password to access the private repository:

```
$ oc secrets new-basicauth basicsecret --username=USERNAME --password=PASSWORD
```

- a. To create a Basic Authentication Secret with a token:

```
$ oc secrets new-basicauth basicsecret --password=TOKEN
```

- b. To create a Basic Authentication Secret with a CA certificate file:

```
$ oc secrets new-basicauth basicsecret --username=USERNAME --password=PASSWORD --ca-cert=FILENAME
```

- c. To create a Basic Authentication Secret with a **.gitconfig** file:

```
$ oc secrets new-basicauth basicsecret --username=USERNAME --password=PASSWORD --gitconfig=FILENAME
```

2. Add the **secret** to the builder service account. Each build is run with **serviceaccount/builder** role, so you need to give it access your secret with following command:

```
$ oc secrets add serviceaccount/builder secrets/basicsecret
```

3. Add a **sourceSecret** field to the **source** section inside the **BuildConfig** and set it to the name of the **secret** that you created. In this case **basicsecret**:

```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  output:
    to:
      kind: "ImageStreamTag"
      name: "sample-image:latest"
  source:
    git:
```

```

    uri: "https://github.com/user/app.git" 1
    sourceSecret:
      name: "basicsecret"
      type: "Git"
    strategy:
      sourceStrategy:
        from:
          kind: "ImageStreamTag"
          name: "python-33-centos7:latest"
        type: "Source"

```

1

The URL of private repository, accessed by basic authentication, is usually in the **http** or **https** form.

11.7.2.2. SSH Key Based Authentication

SSH Key Based Authentication requires a private SSH key. A **.gitconfig** file can also be attached.

The repository keys are usually located in the **\$HOME/.ssh/** directory, and are named **id_dsa.pub**, **id_ecdsa.pub**, **id_ed25519.pub**, or **id_rsa.pub** by default. Generate SSH key credentials with the following command:

```
$ ssh-keygen -t rsa -C "your_email@example.com"
```



Note

Creating a passphrase for the SSH key prevents OpenShift Enterprise from building. When prompted for a passphrase, leave it blank.

Two files are created: the public key and a corresponding private key (one of **id_dsa**, **id_ecdsa**, **id_ed25519**, or **id_rsa**). With both of these in place, consult your source control management (SCM) system's manual on how to upload the public key. The private key will be used to access your private repository.

A **secret** is used to store your keys.

1. Create the **secret** first before using the SSH key to access the private repository:

```
$ oc secrets new-sshauth sshsecret --ssh-privatekey=$HOME/.ssh/id_rsa
```

- a. To create a SSH Based Authentication Secret with a **.gitconfig** file:

```
$ oc secrets new-sshauth sshsecret --ssh-privatekey=$HOME/.ssh/id_rsa --gitconfig=FILENAME
```

2. Add the **secret** to the builder service account. Each build is run with **serviceaccount/builder** role, so you need to give it access your secret with following command:

```
$ oc secrets add serviceaccount/builder secrets/sshsecret
```

3. Add a **sourceSecret** field into the **source** section inside the **BuildConfig** and set it to the name of the **secret** that you created. In this case **sshsecret**:

```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  output:
    to:
      kind: "ImageStreamTag"
      name: "sample-image:latest"
  source:
    git:
      uri: "git@repository.com:user/app.git" 1
      sourceSecret:
        name: "sshsecret"
      type: "Git"
  strategy:
    sourceStrategy:
      from:
        kind: "ImageStreamTag"
        name: "python-33-centos7:latest"
      type: "Source"
```

1

The URL of private repository, accessed by a private SSH key, is usually in the form **git@example.com:<username>/<repository>.git**.

11.7.2.3. Other

If the cloning of your application is dependent on a CA certificate, **.gitconfig** file, or both, then you can create a secret that contains them, add it to the builder service account, and then your **BuildConfig**.

1. Create desired type of **secret**:
 - a. To create a secret from a **.gitconfig**:

-

```
$ oc secrets new mysecret .gitconfig=path/to/.gitconfig
```

- b. To create a secret from a **CA certificate**:

```
$ oc secrets new mysecret ca.crt=path/to/certificate
```

- c. To create a secret from a **CA certificate** and **.gitconfig**:

```
$ oc secrets new mysecret ca.crt=path/to/certificate
.gitconfig=path/to/.gitconfig
```

Note

SSL verification can be turned off, if **sslVerify=false** is set for the **http** section in your **.gitconfig** file:

```
[http]
    sslVerify=false
```

2. Add the **secret** to the builder service account. Each build is run with the **serviceaccount/builder** role, so you need to give it access your secret with following command:

```
$ oc secrets add serviceaccount/builder secrets/mysecret
```

3. Add the **secret** to the **BuildConfig**:

```
source:
  git:
    uri: "https://github.com/openshift/nodejs-ex.git"
  sourceSecret:
    name: "mysecret"
```

[Defining Secrets in the BuildConfig](#) provides more information on this topic.

11.8. DOCKERFILE SOURCE

When the **BuildConfig.spec.source.type** is **Dockerfile**, an inline Dockerfile is used as the build input, and no additional sources can be provided.

This source type is valid when the build strategy type is **Docker** or **Custom**.

The source definition is part of the **spec** section in the **BuildConfig**:

```
source:
  type: "Dockerfile"
  dockerfile: "FROM centos:7\nRUN yum install -y httpd" 1
```

The **dockerfile** field contains an inline Dockerfile that will be built.

11.9. BINARY SOURCE

Streaming content in binary format from a local file system to the builder is called a **binary type build**. The corresponding value of **BuildConfig.spec.source.type** is **Binary** for such builds.

This source type is unique in that it is leveraged solely based on your use of the **oc start-build**.



Note

Binary type builds require content to be streamed from the local file system, so automatically triggering a binary type build (e.g. via an image change trigger) is not possible, because the binary files cannot be provided. Similarly, you cannot launch binary type builds from the web console.

To utilize binary builds, invoke **oc start-build** with one of these options:

- ✎ **--from-file**: The contents of the file you specify are sent as a binary stream to the builder. The builder then stores the data in a file with the same name at the top of the build context.
- ✎ **--from-dir** and **--from-repo**: The contents are archived and sent as a binary stream to the builder. The builder then extracts the contents of the archive within the build context directory.

In each of the above cases:

- ✎ If your **BuildConfig** already has a **Binary** source type defined, it will effectively be ignored and replaced by what the client sends.
- ✎ If your **BuildConfig** has a **Git** source type defined, it is dynamically disabled, since **Binary** and **Git** are mutually exclusive, and the data in the binary stream provided to the builder takes precedence.

When using **oc new-build --binary=true**, the command ensures that the restrictions associated with binary builds are enforced. The resulting **BuildConfig** will have a source type of **Binary**, meaning that the only valid way to run a build for this **BuildConfig** is to use **oc start-build** with one of the **--from** options to provide the requisite binary data.

The **dockerfile** and **contextDir** [source options](#) have special meaning with binary builds.

dockerfile can be used with any binary build source. If **dockerfile** is used and the binary stream is an archive, its contents serve as a replacement Dockerfile to any Dockerfile in the archive. If **dockerfile** is used with the **--from-file** argument, and the file argument is named **dockerfile**, the value from **dockerfile** replaces the value from the binary stream.

In the case of the binary stream encapsulating extracted archive content, the value of the **contextDir** field is interpreted as a subdirectory within the archive, and, if valid, the builder changes into that subdirectory before executing the build.

11.10. IMAGE SOURCE

Additional files can be provided to the build process via images. Input images are referenced in the same way the **From** and **To** image targets are defined. This means both container images and image stream tags can be referenced. In conjunction with the image, you must provide one or more path pairs to indicate the path of the files/directories to copy out of the image and the destination to place them in the build context.

The source path can be any absolute path within the image specified. The destination must be a relative directory path. At build time, the image will be loaded and the indicated files and directories will be copied into the context directory of the build process. This is the same directory into which the source repository content (if any) is cloned. If the source path ends in `/`, then the content of the directory will be copied, but the directory itself will not be created at the destination.

Image inputs are specified in the **source** definition of the **BuildConfig**:

```
source:
  git:
    uri: https://github.com/openshift/ruby-hello-world.git
  images: 1
  - from: 2
    kind: ImageStreamTag
    name: myinputimage:latest
    namespace: mynamespace
    paths: 3
    - destinationDir: injected/dir 4
      sourcePath: /usr/lib/somefile.jar 5
  - from:
    kind: ImageStreamTag
    name: myotherinputimage:latest
    namespace: myothernamespace
    pullSecret: mysecret 6
    paths:
    - destinationDir: injected/dir
      sourcePath: /usr/lib/somefile.jar
```

1

An array of one or more input images and files.

2

A reference to the image containing the files to be copied.

3

An array of source/destination paths.

4

The directory relative to the build root where the build process can access the file.

5

The location of the file to be copied out of the referenced image.

6

An optional secret provided if credentials are needed to access the input image.



Note

This feature is not supported for builds using the [Custom Strategy](#).

11.11. USING SECRETS DURING A BUILD

In some scenarios, build operations require credentials to access dependent resources, but it is undesirable for those credentials to be available in the final application image produced by the build.

For example, when building a NodeJS application, you can set up your private mirror for NodeJS modules. In order to download modules from that private mirror, you have to supply a custom **.npmrc** file for the build that contains a URL, user name, and password. For security reasons, you do not want to expose your credentials in the application image.

This example describes NodeJS, but you can use the same approach for adding SSL certificates into the **/etc/ssl/certs** directory, API keys or tokens, license files, etc.

11.11.1. Defining Secrets in the BuildConfig

1. Create the **Secret**:

```
$ oc secrets new secret-npmrc .npmrc=~/.npmrc
```

This creates a new secret named **secret-npmrc**, which contains the base64 encoded content of the **~/.npmrc** file.

2. Add the secret to the **source** section in the existing build configuration:

```
source:
  git:
    uri: https://github.com/openshift/nodejs-ex.git
  secrets:
    - secret:
        name: secret-npmrc
  type: Git
```


To include the secrets in a new build configuration, run the following command:

```
$ oc new-build openshift/nodejs-010-
centos7~https://github.com/openshift/nodejs-ex.git --build-secret
secret-npmrc
```

During the build, the **.npmrc** file is copied into the directory where the source code is located. In case of the OpenShift Enterprise S2I builder images, this is the image working directory, which is set using the **WORKDIR** instruction in the Dockerfile. If you want to specify another directory, add a **destinationDir** to the secret definition:

```
source:
  git:
    uri: https://github.com/openshift/nodejs-ex.git
  secrets:
    - secret:
        name: secret-npmrc
        destinationDir: /etc
  type: Git
```

You can also specify the destination directory when creating a new build configuration:

```
$ oc new-build openshift/nodejs-010-
centos7~https://github.com/openshift/nodejs-ex.git --build-secret
"secret-npmrc:/etc"
```

In both cases, the **.npmrc** file is added to the **/etc** directory of the build environment. Note that for a [Docker strategy](#) the destination directory must be a relative path.

11.11.2. Source-to-Image Strategy

When using a **Source** strategy, all defined source secrets are copied to their respective **destinationDir**. If you left **destinationDir** empty, then the secrets are placed in the working directory of the builder image. The same rule is used when a **destinationDir** is a relative path; the secrets are placed in the paths that are relative to the image's working directory. The **destinationDir** must exist or an error will occur. No directory paths are created during the copy process.



Note

Currently, any files with these secrets are world-writable (have **0666** permissions) and will be truncated to size zero after executing the **assemble** script. This means that the secret files will exist in the resulting image, but they will be empty for security reasons.

11.11.3. Docker Strategy

When using a **Docker** strategy, you can add all defined source secrets into your container image using the [ADD](#) and [COPY instructions](#) in your **Dockerfile**. If you do not specify the **destinationDir** for a secret, then the files will be copied into the same directory in which the **Dockerfile** is located. If you specify a relative path as **destinationDir**, then the secrets will be

copied into that directory, relative to your **Dockerfile** location. This makes the secret files available to the Docker build operation as part of the context directory used during the build.



Note

Users should always remove their secrets from the final application image so that the secrets are not present in the container running from that image. However, the secrets will still exist in the image itself in the layer where they were added. This removal should be part of the **Dockerfile** itself.

11.11.4. Custom Strategy

When using a **Custom** strategy, then all the defined source secrets are available inside the builder container in the **/var/run/secrets/openshift.io/build** directory. The custom build image is responsible for using these secrets appropriately. The **Custom** strategy also allows secrets to be defined as described in [Secrets](#). There is no technical difference between existing strategy secrets and the source secrets. However, your builder image might distinguish between them and use them differently, based on your build use case. The source secrets are always mounted into the **/var/run/secrets/openshift.io/build** directory or your builder can parse the **\$BUILD** environment variable, which includes the full build object.

11.12. STARTING A BUILD

Manually start a new build from an existing build configuration in your current project using the following command:

```
$ oc start-build <buildconfig_name>
```

Re-run a build using the **--from-build** flag:

```
$ oc start-build --from-build=<build_name>
```

Specify the **--follow** flag to stream the build's logs in stdout:

```
$ oc start-build <buildconfig_name> --follow
```

Specify the **--env** flag to set any desired environment variable for the build:

```
$ oc start-build <buildconfig_name> --env=<key>=<value>
```

Rather than relying on a Git source pull or a Dockerfile for a build, you can also start a build by directly pushing your source, which could be the contents of a Git or SVN working directory, a set of prebuilt binary artifacts you want to deploy, or a single file. This can be done by specifying one of the following options for the **start-build** command:

Option	Description
--from-dir=<directory>	Specifies a directory that will be archived and used as a binary input for the build.
--from-file=<file>	Specifies a single file that will be the only file in the build source. The file is placed in the root of an empty directory with the same file name as the original file provided.
--from-repo=<local_source_repo>	Specifies a path to a local repository to use as the binary input for a build. Add the --commit option to control which branch, tag, or commit is used for the build.

When passing any of these options directly to the build, the contents are streamed to the build and override the current build source settings.



Note

Builds triggered from binary input will not preserve the source on the server, so rebuilds triggered by base image changes will use the source specified in the build configuration.

For example, the following command sends the contents of a local Git repository as an archive from the tag **v2** and starts a build:

```
$ oc start-build hello-world --from-repo=../hello-world --commit=v2
```

11.13. CANCELING A BUILD

Manually cancel a build using the web console, or with the following CLI command:

```
$ oc cancel-build <build_name>
```

11.14. DELETING A BUILDCONFIG

Delete a **BuildConfig** using the following command:

```
$ oc delete bc <BuildConfigName>
```

This will also delete all builds that were instantiated from this **BuildConfig**. Specify the **--cascade=false** flag if you do not want to delete the builds:

```
$ oc delete --cascade=false bc <BuildConfigName>
```

11.15. VIEWING BUILD DETAILS

11.15. VIEWING BUILD DETAILS

You can view build details using the web console or the following CLI command:

```
$ oc describe build <build_name>
```

The output of the **describe** command includes details such as the build source, strategy, and output destination. If the build uses the Docker or Source strategy, it will also include information about the source revision used for the build: commit ID, author, committer, and message.

11.16. ACCESSING BUILD LOGS

You can access build logs using the web console or the CLI.

To stream the logs using the build directly:

```
$ oc logs -f build/<build_name>
```

To stream the logs of the latest build for a build configuration:

```
$ oc logs -f bc/<buildconfig_name>
```

To return the logs of a given version build for a build configuration:

```
$ oc logs --version=<number> bc/<buildconfig_name>
```

Log Verbosity

To enable more verbose output, pass the **BUILD_LOGLEVEL** environment variable as part of the **sourceStrategy** or **dockerStrategy** in a **BuildConfig**:

```
sourceStrategy:
...
env:
- name: "BUILD_LOGLEVEL"
  value: "2" 1
```

1

Adjust this value to the desired log level.



Note

A platform administrator can set verbosity for the entire OpenShift Enterprise instance by passing the **--loglevel** option to the **openshift start** command. If both **--loglevel** and **BUILD_LOGLEVEL** are specified, **BUILD_LOGLEVEL** takes precedence.

Available log levels for Source builds are as follows:

Level 0	Produces output from containers running the assemble script and all encountered errors. This is the default.
Level 1	Produces basic information about the executed process.
Level 2	Produces very detailed information about the executed process.
Level 3	Produces very detailed information about the executed process, and a listing of the archive contents.
Level 4	Currently produces the same information as level 3.
Level 5	Produces everything mentioned on previous levels and additionally provides docker push messages.

11.17. SETTING MAXIMUM DURATION

When defining a **BuildConfig**, you can define its maximum duration by setting the **completionDeadlineSeconds** field. It is specified in seconds and is not set by default. When not set, there is no maximum duration enforced.

The maximum duration is counted from the time when a build pod gets scheduled in the system, and defines how long it can be active, including the time needed to pull the builder image. After reaching the specified timeout, the build is terminated by OpenShift Enterprise.

The following example shows the part of a **BuildConfig** specifying **completionDeadlineSeconds** field for 30 minutes:

```
spec:
  completionDeadlineSeconds: 1800
```

11.18. BUILD TRIGGERS

When defining a **BuildConfig**, you can define triggers to control the circumstances in which the **BuildConfig** should be run. The following build triggers are available:

- 🔗 [Webhook](#)
- 🔗 [Image change](#)
- 🔗 [Configuration change](#)

11.18.1. Webhook Triggers

Webhook triggers allow you to trigger a new build by sending a request to the OpenShift Enterprise API endpoint. You can define these triggers using [GitHub webhooks](#) or Generic webhooks.

GitHub Webhooks

[GitHub webhooks](#) handle the call made by GitHub when a repository is updated. When defining the trigger, you must specify a **secret**, which will be part of the URL you supply to GitHub when configuring the webhook. The secret ensures the uniqueness of the URL, preventing others from triggering the build. The following example is a trigger definition YAML within the **BuildConfig**:

```
type: "GitHub"
github:
  secret: "secret101"
```

Note

The secret field in webhook trigger configuration is not the same as **secret** field you encounter when configuring webhook in GitHub UI. The former is to make the webhook URL unique and hard to predict, the latter is an optional string field used to create HMAC hex digest of the body, which is sent as an **X-Hub-Signature** header.

The payload URL is returned as the GitHub Webhook URL by the **describe** command (see [below](#)), and is structured as follows:

```
http://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildco
nfigs/<name>/webhooks/<secret>/github
```

To configure a GitHub Webhook:

1. Describe the build configuration to get the webhook URL:

```
$ oc describe bc <name>
```

2. Copy the webhook URL.
3. Follow the [GitHub setup instructions](#) to paste the webhook URL into your GitHub repository settings.

Note

[Gogs](#) supports the same webhook payload format as GitHub. Therefore, if you are using a Gogs server, you can define a GitHub webhook trigger on your **BuildConfig** and trigger it via your Gogs server also.

Generic Webhooks

Generic webhooks can be invoked from any system capable of making a web request. As with a GitHub webhook, you must specify a **secret** which will be part of the URL, the caller must use to trigger the build. The secret ensures the uniqueness of the URL, preventing others from triggering the build. The following is an example trigger definition YAML within the **BuildConfig**:

```
type: "Generic"
generic:
  secret: "secret101"
```

To set up the caller, supply the calling system with the URL of the generic webhook endpoint for your build:

```
http://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildco
nfigs/<name>/webhooks/<secret>/generic
```

The endpoint can accept an optional payload with the following format:

```
type: "git"
git:
  uri: "<url to git repository>"
  ref: "<optional git reference>"
  commit: "<commit hash identifying a specific git commit>"
  author:
    name: "<author name>"
    email: "<author e-mail>"
  committer:
    name: "<committer name>"
    email: "<committer e-mail>"
  message: "<commit message>"
```

Displaying a BuildConfig's Webhook URLs

Use the following command to display the webhook URLs associated with a build configuration:

```
$ oc describe bc <name>
```

If the above command does not display any webhook URLs, then no webhook trigger is defined for that build configuration.

11.18.2. Image Change Triggers

Image change triggers allow your build to be automatically invoked when a new version of an upstream image is available. For example, if a build is based on top of a RHEL image, then you can trigger that build to run any time the RHEL image changes. As a result, the application image is always running on the latest RHEL base image.

Configuring an image change trigger requires the following actions:

1. Define an **ImageStream** that points to the upstream image you want to trigger on:

```
kind: "ImageStream"
apiVersion: "v1"
metadata:
  name: "ruby-20-centos7"
```

This defines the image stream that is tied to a container image repository located at **<system-registry>/<namespace>/ruby-20-centos7**. The **<system-registry>** is defined as a service with the name **docker-registry** running in OpenShift Enterprise.

2. If an image stream is the base image for the build, set the **from** field in the build strategy to point to the image stream:

```
strategy:
  type: "Source"
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "ruby-20-centos7:latest"
```

In this case, the **sourceStrategy** definition is consuming the **latest** tag of the image stream named **ruby-20-centos7** located within this namespace.

3. Define a build with one or more triggers that point to image streams:

```
type: "imageChange" 1
imageChange: {}
type: "imagechange" 2
imageChange:
  from:
    kind: "ImageStreamTag"
    name: "custom-image:latest"
```

1

An image change trigger that monitors the **ImageStream** and **Tag** as defined by the build strategy's **from** field. The **imageChange** object here must be empty.

2

An image change trigger that monitors an arbitrary image stream. The **imageChange** part in this case must include a **from** field that references the **ImageStreamTag** to monitor.

When using an image change trigger for the strategy image stream, the generated build is supplied with an immutable Docker tag that points to the latest image corresponding to that tag. This new image reference will be used by the strategy when it executes for the build. For other image change triggers that do not reference the strategy image stream, a new build will be started, but the build strategy will not be updated with a unique image reference.

In the example above that has an image change trigger for the strategy, the resulting build will be:

```
strategy:
  type: "Source"
  sourceStrategy:
    from:
```



```
kind: "DockerImage"
name: "172.30.17.3:5001/mynamespace/ruby-20-centos7:immutableid"
```

This ensures that the triggered build uses the new image that was just pushed to the repository, and the build can be re-run any time with the same inputs.

In addition to setting the image field for all **Strategy** types, for custom builds, the **OPENSIFT_CUSTOM_BUILD_BASE_IMAGE** environment variable is checked. If it does not exist, then it is created with the immutable image reference. If it does exist then it is updated with the immutable image reference.

If a build is triggered due to a webhook trigger or manual request, the build that is created uses the **immutableid** resolved from the **ImageStream** referenced by the **Strategy**. This ensures that builds are performed using consistent image tags for ease of reproduction.



Note

Image streams that point to container images in [v1 Docker registries](#) only trigger a build once when the image stream tag becomes available and not on subsequent image updates. This is due to the lack of uniquely identifiable images in v1 Docker registries.

11.18.3. Configuration Change Triggers

A configuration change trigger allows a build to be automatically invoked as soon as a new **BuildConfig** is created. The following is an example trigger definition YAML within the **BuildConfig**:

```
type: "ConfigChange"
```



Note

Configuration change triggers currently only work when creating a new **BuildConfig**. In a future release, configuration change triggers will also be able to launch a build whenever a **BuildConfig** is updated.

11.19. BUILD HOOKS

Build hooks allow behavior to be injected into the build process.

Use the **postCommit** field to execute commands inside a temporary container that is running the build output image. The hook is executed immediately after the last layer of the image has been committed and before the image is pushed to a registry.

The current working directory is set to the image's **WORKDIR**, which is the default working directory of the container image. For most images, this is where the source code is located.

The hook fails if the script or command returns a non-zero exit code or if starting the temporary container fails. When the hook fails it marks the build as failed and the image is not pushed to a registry. The reason for failing can be inspected by looking at the build logs.

Build hooks can be used to run unit tests to verify the image before the build is marked complete and

the image is made available in a registry. If all tests pass and the test runner returns with exit code 0, the build is marked successful. In case of any test failure, the build is marked as failed. In all cases, the build log will contain the output of the test runner, which can be used to identify failed tests.

The **postCommit** hook is not only limited to running tests, but can be used for other commands as well. Since it runs in a temporary container, changes made by the hook do not persist, meaning that the hook execution cannot affect the final image. This behavior allows for, among other uses, the installation and usage of test dependencies that are automatically discarded and will be not present in the final image.

There are different ways to configure the post build hook. All forms in the following examples are equivalent and execute **bundle exec rake test --verbose**:

✎ Shell script:

```
postCommit:
  script: "bundle exec rake test --verbose"
```

The **script** value is a shell script to be run with **/bin/sh -ic**. Use this when a shell script is appropriate to execute the build hook. For example, for running unit tests as above. To control the image entry point, or if the image does not have **/bin/sh**, use **command** and/or **args**.



Note

The additional **-i** flag was introduced to improve the experience working with CentOS and RHEL images, and may be removed in a future release.

✎ Command as the image entry point:

```
postCommit:
  command: ["/bin/bash", "-c", "bundle exec rake test --verbose"]
```

In this form, **command** is the command to run, which overrides the image entry point in the exec form, as documented in the [Dockerfile reference](#). This is needed if the image does not have **/bin/sh**, or if you do not want to use a shell. In all other cases, using **script** might be more convenient.

✎ Pass arguments to the default entry point:

```
postCommit:
  args: ["bundle", "exec", "rake", "test", "--verbose"]
```

In this form, **args** is a list of arguments that are provided to the default entry point of the image. The image entry point must be able to handle arguments.

✎ Shell script with arguments:

```
postCommit:
  script: "bundle exec rake test $1"
  args: ["--verbose"]
```

Use this form if you need to pass arguments that would otherwise be hard to quote properly in the shell script. In the **script**, **\$0** will be `/bin/sh` and **\$1**, **\$2**, etc, are the positional arguments from **args**.

❖ Command with arguments:

```
postCommit:
  command: ["bundle", "exec", "rake", "test"]
  args: ["--verbose"]
```

This form is equivalent to appending the arguments to **command**.



Note

Providing both **script** and **command** simultaneously creates an invalid build hook.

11.19.1. Using the Command Line

The **oc set build-hook** command can be used to set the build hook for a build configuration.

To set a command as the post-commit build hook:

```
$ oc set build-hook bc/mybc --post-commit --command -- bundle exec rake
test --verbose
```

To set a script as the post-commit build hook:

```
$ oc set build-hook bc/mybc --post-commit --script="bundle exec rake
test --verbose"
```

11.20. USING DOCKER CREDENTIALS FOR PUSHING AND PULLING IMAGES

Supply the **.dockercfg** file with valid Docker Registry credentials in order to push the output image into a private Docker Registry or pull the builder image from the private Docker Registry that requires authentication. For the OpenShift Enterprise Docker Registry, you don't have to do this because **secrets** are generated automatically for you by OpenShift Enterprise.

The **.dockercfg** JSON file is found in your home directory by default and has the following format:

```
auths:
  https://index.docker.io/v1/: 1
    auth: "YWRfbGZhcGU6R2labnRib21ifTE=" 2
    email: "user@example.com" 3
```

URL of the registry.

2

Encrypted password.

3

Email address for the login.

You can define multiple Docker registry entries in this file. Alternatively, you can also add authentication entries to this file by running the **docker login** command. The file will be created if it does not exist. Kubernetes provides **secret** objects, which are used to store your configuration and passwords.

1. Create the **secret** from your local **.dockercfg** file:

```
$ oc secrets new dockerhub ~/.dockercfg
```

This generates a JSON specification of the **secret** named **dockerhub** and creates the object.

2. Once the **secret** is created, add it to the builder service account. Each build is run with **serviceaccount/builder** role, so you need to give it access your secret with following command:

```
$ oc secrets add serviceaccount/builder secrets/dockerhub
```

3. Add a **pushSecret** field into the **output** section of the **BuildConfig** and set it to the name of the **secret** that you created, which in the above example is **dockerhub**:

```
spec:
  output:
    to:
      kind: "DockerImage"
      name: "private.registry.com/org/private-image:latest"
    pushSecret:
      name: "dockerhub"
```

4. Pull the builder container image from a private Docker registry by specifying the **pullSecret** field, which is part of the build strategy definition:

```
strategy:
  sourceStrategy:
    from:
      kind: "DockerImage"
```

```

    name: "docker.io/user/private_repository"
  pullSecret:
    name: "dockerhub"
  type: "Source"

```



Note

This example uses **pullSecret** in a Source build, but it is also applicable in Docker and Custom builds.

11.21. BUILD RUN POLICY

The build run policy describes the order in which the builds created from the build configuration should run. This can be done by changing the value of the **runPolicy** field in the **spec** section of the **Build** specification.

It is also possible to change the **runPolicy** value for existing build configurations.

- ✳ Changing **Parallel** to **Serial** or **SerialLatestOnly** and triggering a new build from this configuration will cause the new build to wait until all parallel builds complete as the serial build can only run alone.
- ✳ Changing **Serial** to **SerialLatestOnly** and triggering a new build will cause cancellation of all existing builds in queue, except the currently running build and the most recently created build. The newest build will execute next.

11.21.1. Serial Run Policy

Setting the **runPolicy** field to **Serial** will cause all new builds created from the **Build** configuration to be run sequentially. That means there will be only one build running at a time and every new build will wait until the previous build completes. Using this policy will result in consistent and predictable build output. This is the default **runPolicy**.

Triggering three builds from the **sample-build** configuration, using the **Serial** policy will result in:

NAME	TYPE	FROM	STATUS	STARTED
DURATION				
sample-build-1	Source	Git@e79d887	Running	13 seconds ago
13s				
sample-build-2	Source	Git	New	
sample-build-3	Source	Git	New	

When the **sample-build-1** build completes, the **sample-build-2** build will run:

NAME	TYPE	FROM	STATUS	STARTED
DURATION				
sample-build-1	Source	Git@e79d887	Completed	43 seconds ago
34s				
sample-build-2	Source	Git@1aa381b	Running	2 seconds ago
sample-build-3	Source	Git	New	2s

11.21.2. SerialLatestOnly Run Policy

Setting the **runPolicy** field to **SerialLatestOnly** will cause all new builds created from the **Build** configuration to be run sequentially, same as using the **Serial** run policy. The difference is that when a currently running build completes, the next build that will run is the latest build created. In other words, you do not wait for the queued builds to run, as they are skipped. Skipped builds are marked as **Cancelled**. This policy can be used for fast, iterative development.

Triggering three builds from the **sample-build** configuration, using the **SerialLatestOnly** policy will result in:

NAME	TYPE	FROM	STATUS	STARTED
DURATION				
sample-build-1	Source	Git@e79d887	Running	13 seconds ago
13s				
sample-build-2	Source	Git	Cancelled	
sample-build-3	Source	Git	New	

The **sample-build-2** build will be canceled (skipped) and the next build run after **sample-build-1** completes will be the **sample-build-3** build:

NAME	TYPE	FROM	STATUS	STARTED
DURATION				
sample-build-1	Source	Git@e79d887	Completed	43 seconds ago
34s				
sample-build-2	Source	Git	Cancelled	
sample-build-3	Source	Git@1aa381b	Running	2 seconds ago 2s

11.21.3. Parallel Run Policy

Setting the **runPolicy** field to **Parallel** causes all new builds created from the **Build** configuration to be run in parallel. This can produce unpredictable results, as the first created build can complete last, which will replace the pushed container image produced by the last build which completed earlier.

Use the parallel run policy in cases where you do not care about the order in which the builds will complete.

Triggering three builds from the **sample-build** configuration, using the **Parallel** policy will result in three simultaneous builds:

NAME	TYPE	FROM	STATUS	STARTED
DURATION				
sample-build-1	Source	Git@e79d887	Running	13 seconds ago
13s				
sample-build-2	Source	Git@a76d881	Running	15 seconds ago 3s
sample-build-3	Source	Git@689d111	Running	17 seconds ago 3s

The completion order is not guaranteed:

NAME	TYPE	FROM	STATUS	STARTED
------	------	------	--------	---------

DURATION

sample-build-1	Source	Git@e79d887	Running	13 seconds ago	
13s					
sample-build-2	Source	Git@a76d881	Running	15 seconds ago	3s
sample-build-3	Source	Git@689d111	Completed	17 seconds ago	5s

11.22. BUILD OUTPUT

Docker and Source builds result in the creation of a new container image. The image is then pushed to the registry specified in the **output** section of the **Build** specification.

If the output kind is **ImageStreamTag**, then the image will be pushed to the integrated OpenShift Enterprise registry and tagged in the specified image stream. If the output is of type **DockerImage**, then the name of the output reference will be used as a Docker push specification. The specification may contain a registry or will default to DockerHub if no registry is specified. If the output section of the build specification is empty, then the image will not be pushed at the end of the build.

Example 11.2. Output to an ImageStreamTag

```
output:
  to:
    kind: "ImageStreamTag"
    name: "sample-image:latest"
```

Example 11.3. Output to a Docker Push Specification

```
output:
  to:
    kind: "DockerImage"
    name: "my-registry.mycompany.com:5000/myimages/myimage:tag"
```

11.22.1. Output Image Environment Variables

Docker and Source builds set the following environment variables on output images:

Variable	Description
OPENSIFT_BUILD_NAME	Name of the build
OPENSIFT_BUILD_NAMESPACE	Namespace of the build

Variable	Description
OPENSIFT_BUILD_SOURCE	The source URL of the build
OPENSIFT_BUILD_REFERENCE	The Git reference used in the build
OPENSIFT_BUILD_COMMIT	Source commit used in the build

11.22.2. Output Image Labels

Docker and Source builds set the following labels on output images:

Label	Description
io.openshift.build.commit.author	Author of the source commit used in the build
io.openshift.build.commit.date	Date of the source commit used in the build
io.openshift.build.commit.id	Hash of the source commit used in the build
io.openshift.build.commit.message	Message of the source commit used in the build
io.openshift.build.commit.ref	Branch or reference specified in the source
io.openshift.build.source-location	Source URL for the build

11.23. USING EXTERNAL ARTIFACTS DURING A BUILD

It is not recommended to store binary files in a source repository. Therefore, you may find it necessary to define a build which pulls additional files (such as Java *.jar* dependencies) during the build process. How this is done depends on the build strategy you are using.

For a **Source** build strategy, you must put appropriate shell commands into the ***assemble*** script:

Example 11.4. *.s2i/bin/assemble* File

■


```
#!/bin/sh
APP_VERSION=1.0
wget http://repository.example.com/app/app-$APP_VERSION.jar -O app.jar
```

Example 11.5. `.s2i/bin/run` File

```
#!/bin/sh
exec java -jar app.jar
```



Note

For more information on how to control which **assemble** and **run** script is used by a Source build, see [Overriding Builder Image Scripts](#).

For a **Docker** build strategy, you must modify the **Dockerfile** and invoke shell commands with the **RUN** instruction:

Example 11.6. Excerpt of `Dockerfile`

```
FROM jboss/base-jdk:8

ENV APP_VERSION 1.0
RUN wget http://repository.example.com/app/app-$APP_VERSION.jar -O
app.jar

EXPOSE 8080
CMD [ "java", "-jar", "app.jar" ]
```

In practice, you may want to use an environment variable for the file location so that the specific file to be downloaded can be customized using an environment variable defined on the **BuildConfig**, rather than updating the **assemble** script or **Dockerfile**.

You can choose between different methods of defining environment variables:

- ✳ [Using the `.s2i/environment` file](#) (only for a **Source** build strategy)
- ✳ [Setting in `BuildConfig`](#)
- ✳ [Providing explicitly using `oc start-build --env`](#) (only for builds that are triggered manually)

11.24. BUILD RESOURCES

By default, builds are completed by pods using unbound resources, such as memory and CPU. These resources can be limited by specifying resource limits in a project's default container limits.

You can also limit resource use by specifying resource limits as part of the build configuration. In the following example, each of the **resources**, **cpu**, and **memory** parameters are optional:

```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  resources:
    limits:
      cpu: "100m" 1
      memory: "256Mi" 2
```

1

cpu is in CPU units: **100m** represents 0.1 CPU units ($100 * 1e-3$).

2

memory is in bytes: **256Mi** represents 268435456 bytes ($256 * 2^{20}$).

However, if a [quota](#) has been defined for your project, one of the following two items is required:

- ✦ A **resources** section set with an explicit **requests**:

```
resources:
  requests: 1
    cpu: "100m"
    memory: "256Mi"
```

1

The **requests** object contains the list of resources that correspond to the list of resources in the quota.

- ✦ A [limit range](#) defined in your project, where the defaults from the **LimitRange** object apply to pods created during the build process.

Otherwise, build pod creation will fail, citing a failure to satisfy quota.

11.25. TROUBLESHOOTING

Table 11.1. Troubleshooting Guidance for Builds

Issue	Resolution
A build fails with: <pre>requested access to the resource is denied</pre>	You have exceeded one of the image quotas set on your project. Check your current quota and verify the limits applied and storage in use: <pre>\$ oc describe quota</pre>

CHAPTER 12. MANAGING IMAGES

12.1. OVERVIEW

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

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

There are many ways you can interact with images and set up image streams, depending on where the images' registries are located, any authentication requirements around those registries, and how you want your builds and deployments to behave. The following sections cover a range of these topics.

12.2. TAGGING IMAGES

Before working with OpenShift Enterprise image streams and their tags, it will help to first understand image tags in the context of Docker generally.

Container images can have names added to them that make it more intuitive to determine what they contain, called a *tag*. Using a tag to specify the version of what is contained in the image is a common use case. If you have an image named **ruby**, you could have a tag named **2.0** for 2.0 version of Ruby, and another named **latest** to indicate literally the latest built image in that repository overall.

When interacting directly with images using the **docker** CLI, the **docker tag** command can add tags, which essentially adds an alias to an image that can consist of several parts. Those parts can include:

```
<registry_server>/<user_name>/<image_name>:<tag>
```

The **<user_name>** part in the above could also refer to a [project](#) or [namespace](#) if the image is being stored in an OpenShift Enterprise environment with an internal registry.

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



Note

See Red Hat Enterprise Linux 7's [Getting Started with Containers](#) documentation for more about tagging images directly using the **docker** CLI.

12.2.1. Adding Tags to Image Streams

Keeping in mind that an image stream in OpenShift Enterprise comprises zero or more container images identified by tags, you can add tags to an image stream using the **oc tag** command:

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

For example, to configure the **ruby** image's **latest** tag to always refer to the current image for the tag **2.0**:

```
$ oc tag ruby:latest ruby:2.0
```

There are different types of tags available. The default behavior uses a *permanent* tag, which points to a specific image in time; even when the source changes, it will not reflect in the destination tag.

A *tracking* tag means the destination tag's metadata will be imported during the import. To ensure the destination tag is updated whenever the source tag changes, use the **--alias=true** flag:

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

You can also add the **--scheduled=true** flag to have the destination tag be refreshed (i.e., re-imported) periodically. The period is configured globally at system level. See [Importing Tag and Image Metadata](#) for more details.



Important

Avoid tagging OpenShift Enterprise-managed images (i.e., those built using an OpenShift Enterprise instance and pushed to its internal registry). There is a [known issue](#) that prevents the registry client from pulling from such a tag.

12.2.2. Removing Tags from Image Streams

You can stop tracking a tag by removing the tag. For example, to stop tracking an existing **latest** tag:

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

However, while the above command removes the tag from the image stream definition, it does not remove it from the image stream status. The image stream definition is user-defined, whereas the image stream status reflects the information the system has from the specification.

To remove a tag completely from an image stream:

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

12.2.3. Referencing Images in Image Streams

Images can be referenced in image streams using the following reference types:

- ✱ An **ImageStreamTag** is used to reference or retrieve an image for a given image stream and tag. It uses the following convention for its name:

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

- ✱ An **ImageStreamImage** is used to reference or retrieve an image for a given image stream and image name. It uses the following convention for its name:

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

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

- ✱ A **DockerImage** is used to reference or retrieve an image for a given external registry. It uses standard Docker *pull specification* for its name, e.g.:

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



Note

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

You can also reference a third-party registry:

```
registry.access.redhat.com/rhel7:latest
```

Or an image with a digest:

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

When viewing example image stream definitions, such as the [example CentOS image streams](#), 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 Enterprise whenever you import or tag an image into the image stream. You should never have to explicitly define an **ImageStreamImage** object in any image stream definition that you use to create image streams.

You can view an image's object definition by retrieving an **ImageStreamImage** definition using the image stream name and ID:

```
$ oc export isimage <image_stream_name>@<id>
```



Note

You can find valid **<id>** values for a given image stream by running:

```
$ oc describe is <image_stream_name>
```

For example, from the **ruby** image stream asking for the **ImageStreamImage** with the name and ID of **ruby@3a335d7**:

Example 12.1. Definition of an Image Object Retrieved via ImageStreamImage

```
$ oc export isimage ruby@3a335d7

apiVersion: v1
image:
  dockerImageLayers:
```

```

- name:
sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46
d4
  size: 0
- name:
sha256:ee1dd2cb6df21971f4af6de0f1d7782b81fb63156801cfde2bb47b4247c23c
29
  size: 196634330
- name:
sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46
d4
  size: 0
- name:
sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46
d4
  size: 0
- name:
sha256:ca062656bff07f18bff46be00f40cfbb069687ec124ac0aa038fd676cfaea0
92
  size: 177723024
- name:
sha256:63d529c59c92843c395befd065de516ee9ed4995549f8218eac6ff088bfa6b
6e
  size: 55679776
dockerImageMetadata:
  Architecture: amd64
  Author: SoftwareCollections.org <sclorg@redhat.com>
  Config:
    Cmd:
      - /bin/sh
      - -c
      - $STI_SCRIPTS_PATH/usage
    Entrypoint:
      - container-entrypoint
    Env:
      - PATH=/opt/app-root/src/bin:/opt/app-
root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
      - STI_SCRIPTS_URL=image:///usr/libexec/s2i
      - STI_SCRIPTS_PATH=/usr/libexec/s2i
      - HOME=/opt/app-root/src
      - BASH_ENV=/opt/app-root/etc/scl_enable
      - ENV=/opt/app-root/etc/scl_enable
      - PROMPT_COMMAND=. /opt/app-root/etc/scl_enable
      - RUBY_VERSION=2.2
    ExposedPorts:
      8080/tcp: {}
    Image:
d9c3abc5456a9461954ff0de8ae25e0e016aad35700594714d42b687564b1f51
    Labels:
      build-date: 2015-12-23
      io.k8s.description: Platform for building and running Ruby
2.2 applications
      io.k8s.display-name: Ruby 2.2
      io.openshift.builder-base-version: 8d95148
      io.openshift.builder-version:
8847438ba06307f86ac877465eadc835201241df

```

```

    io.openshift.expose-services: 8080:http
    io.openshift.s2i.scripts-url: image:///usr/libexec/s2i
    io.openshift.tags: builder,ruby,ruby22
    io.s2i.scripts-url: image:///usr/libexec/s2i
    license: GPLv2
    name: CentOS Base Image
    vendor: CentOS
    User: "1001"
    WorkingDir: /opt/app-root/src
    ContainerConfig: {}
    Created: 2016-01-26T21:07:27Z
    DockerVersion: 1.8.2-e17
    Id:
57b08d979c86f4500dc8cad639c9518744c8dd39447c055a3517dc9c18d6fccd
    Parent:
d9c3abc5456a9461954ff0de8ae25e0e016aad35700594714d42b687564b1f51
    Size: 430037130
    apiVersion: "1.0"
    kind: DockerImage
    dockerImageMetadataVersion: "1.0"
    dockerImageReference: centos/ruby-22-
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c74
6e8986b28e
    metadata:
      creationTimestamp: 2016-01-29T13:17:45Z
      name:
sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b2
8e
      resourceVersion: "352"
      uid: af2e7a0c-c68a-11e5-8a99-525400f25e34
    kind: ImageStreamImage
    metadata:
      creationTimestamp: null
      name: ruby@3a335d7
      namespace: openshift
      selflink:
/oapi/v1/namespaces/openshift/imagestreamimages/ruby@3a335d7

```

12.3. 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.

When OpenShift Enterprise 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**:

- ✳ **Always** - always pull the image.
- ✳ **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 Enterprise sets it based on the image's tag:

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

12.4. ACCESSING THE INTERNAL REGISTRY

You can access OpenShift Enterprise's internal registry directly to push or pull images. For example, this could be helpful if you wanted to [create an image stream by manually pushing an image](#), or just to **docker pull** an image directly.

The internal registry authenticates using the same [tokens](#) as the OpenShift Enterprise API. To perform a **docker login** against the internal registry, you can choose any user name and email, but the password must be a valid OpenShift Enterprise token.

To log into the internal registry:

1. Log in to OpenShift Enterprise:

```
$ oc login
```

2. Get your access token:

```
$ oc whoami -t
```

3. Log in to the internal registry using the token. You must have **docker** installed on your system:

```
$ docker login -u <user_name> -e <email_address> \
  -p <token_value> <registry_server>:<port>
```



Note

Contact your cluster administrator if you do not know the registry IP or host name and port to use.

In order to pull an image, the authenticated user must have **get** rights on the requested **imagestreams/layers**. In order to push an image, the authenticated user must have **update** rights on the requested **imagestreams/layers**.

By default, all service accounts in a project have rights to pull any image in the same project, and the **builder** service account has rights to push any image in the same project.

12.5. USING IMAGE PULL SECRETS

[Docker registries](#) can be secured to prevent unauthorized parties from accessing certain images. If you are [using OpenShift Enterprise's internal registry](#) and are pulling from image streams 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 Enterprise projects or from secured registries, then additional configuration steps are required. The following sections detail these scenarios and their required steps.

12.5.1. 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**:

```
$ 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 will be able to pull images from **project-b**.

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
```

12.5.2. Allowing Pods to Reference Images from Other Secured Registries

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

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

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

```
$ oc secrets new <pull_secret_name> .dockercfg=<path/to/.dockercfg>
```

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

```
$ oc secrets new <pull_secret_name> .dockerconfigjson=
<path/to/.docker/config.json>
```

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

```
$ oc secrets new-dockercfg <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 will use; **default** is the default service account:

```
$ oc secrets add serviceaccount/default secrets/<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 add serviceaccount/builder secrets/<pull_secret_name>
```

12.6. IMPORTING TAG AND IMAGE METADATA

An image stream can be configured to import tag and image metadata from an image repository in an external Docker image registry. You can do this using a few different methods.

- You can manually import tag and image information with the **oc import-image** command using the **--from** option:

```
$ oc import-image <image_stream_name>[:<tag>] --from=
<docker_image_repo> --confirm
```

For example:

```
$ oc import-image my-ruby --from=docker.io/openshift/ruby-20-centos7
--confirm
The import completed successfully.
```

```
Name:      my-ruby
Created:   Less than a second ago
Labels:    <none>
Annotations: openshift.io/image.dockerRepositoryCheck=2016-05-
06T20:59:30Z
Docker Pull Spec: 172.30.94.234:5000/demo-project/my-ruby

Tag Spec      Created      PullSpec      Image
latest docker.io/openshift/ruby-20-centos7 Less than a second ago
docker.io/openshift/ruby-20-centos7@sha256:772c5bf9b2d1e8... <same>
```

You can also add the **--all** flag to import all tags for the image instead of just **latest**.

- Like most objects in OpenShift Enterprise, you can also write and save a JSON or YAML definition to a file then create the object using the CLI. Set the **spec.dockerImageRepository** field to the Docker pull spec for the image:

```
apiVersion: "v1"
kind: "ImageStream"
metadata:
  name: "my-ruby"
spec:
  dockerImageRepository: "docker.io/openshift/ruby-20-centos7"
```

Then create the object:

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

When you create an image stream that references an image in an external Docker registry, OpenShift Enterprise communicates with the external registry within a short amount of time to get up to date information about the image.

After the tag and image metadata is synchronized, the image stream object would look similar to the following:

```
apiVersion: v1
kind: ImageStream
metadata:
  name: my-ruby
  namespace: demo-project
  selflink: /oapi/v1/namespaces/demo-project/imagestreams/my-ruby
  uid: 5b9bd745-13d2-11e6-9a86-0ada84b8265d
  resourceVersion: '4699413'
  generation: 2
  creationTimestamp: '2016-05-06T21:34:48Z'
  annotations:
    openshift.io/image.dockerRepositoryCheck: '2016-05-06T21:34:48Z'
spec:
  dockerImageRepository: docker.io/openshift/ruby-20-centos7
  tags:
  -
    name: latest
    annotations: null
    from:
      kind: DockerImage
      name: 'docker.io/openshift/ruby-20-centos7:latest'
      generation: 2
      importPolicy: { }
status:
  dockerImageRepository: '172.30.94.234:5000/demo-project/my-ruby'
  tags:
  -
    tag: latest
    items:
    -
      created: '2016-05-06T21:34:48Z'
      dockerImageReference: 'docker.io/openshift/ruby-20-centos7@sha256:772c5bf9b2d1e8e80742ed75aab05820419dc4532fa6d7ad8a1efddd
a5493dc3'
      image:
        'sha256:772c5bf9b2d1e8e80742ed75aab05820419dc4532fa6d7ad8a1efddda5493dc
3'
      generation: 2
```

You can set a tag to query external registries at a scheduled interval to synchronize tag and image metadata by setting the `--scheduled=true` flag with the `oc tag` command as mentioned in [Adding Tags to Image Streams](#).

Alternatively, you can set `importPolicy.scheduled` to `true` in the tag's definition:

```
apiVersion: v1
kind: ImageStream
metadata:
```

```

name: ruby
spec:
  tags:
  - from:
      kind: DockerImage
      name: openshift/ruby-20-centos7
    name: latest
    importPolicy:
      scheduled: true

```

12.6.1. Importing Images from Insecure Registries

An image stream can be configured to import tag and image metadata from insecure image registries, such as those signed with a self-signed certificate or using plain HTTP instead of HTTPS.

To configure this, add the **openshift.io/image.insecureRepository** annotation and set it to **true**. This setting bypasses certificate validation when connecting to the registry:

```

kind: ImageStream
apiVersion: v1
metadata:
  name: ruby
  annotations:
    openshift.io/image.insecureRepository: "true"
spec:
  dockerImageRepository: my.repo.com:5000/myimage

```

1

Set the **openshift.io/image.insecureRepository** annotation to **true**

Important

The above definition only affects importing tag and image metadata. For this image to be used in the cluster (e.g., to be able to do a **docker pull**), each node must have Docker configured with the **--insecure-registry** flag. See [Host Preparation](#) for information.

Additionally, you can specify a single tag using an insecure repository. To do so, set **importPolicy.insecure** in the tag's definition to **true**:

```

kind: ImageStream
apiVersion: v1
metadata:
  name: ruby
  tags:
  - from:
      kind: DockerImage

```

```
name: my.repo.com:5000/myimage
name: mytag
importPolicy:
  insecure: true 1
```

1

Set tag **mytag** to use insecure connection to that registry.

12.6.2. Importing Images from Private Registries

An image stream can be configured to import tag and image metadata from private image registries, requiring authentication.

To configure this, you need to create a [secret](#) which is used to store your credentials.

Create the secret first, before importing the image from the private repository:

```
$ oc secrets new-dockercfg <secret_name> \
  --docker-server=<docker_registry_server> \
  --docker-username=<docker_user> \
  --docker-password=<docker_password> \
  --docker-email=<docker_email>
```

For more options, see:

```
$ oc secrets new-dockercfg --help
```

After the secret is configured, proceed with creating the new image stream or using the **oc import-image** command. During the import process, OpenShift Enterprise will pick up the secrets and provide them to the remote party.

12.6.3. Importing Images Across Projects

An image stream can be configured to import tag and image metadata from the internal registry, but from a different project. The recommended method for this is to use the **oc tag** command as shown in [Adding Tags to Image Streams](#):

```
$ oc tag <source_project>/<image_stream>:<tag> <new_image_stream>:
<new_tag>
```

Another method is to import the image from the other project manually using the pull spec:

Warning

The following method is strongly discouraged and should be used only if the former using **oc tag** is insufficient.

1. First, add the necessary [policy](#) to access the other project:

```
$ oc policy add-role-to-group \
  system:image-puller \
  system:serviceaccounts:<destination_project> \
  -n <source_project>
```

This allows **<destination_project>** to pull images from **<source_project>**.

2. With the policy in place, you can import the image manually:

```
$ oc import-image <new_image_stream> --confirm \
  --from=<docker_registry>/<source_project>/<image_stream>
```

12.6.4. Creating an Image Stream by Manually Pushing an Image

An image stream can also be automatically created by manually pushing an image to the internal registry. This is only possible when using an OpenShift Enterprise internal registry.

Before performing this procedure, the following must be satisfied:

- ✦ The destination project you push to must already exist.
- ✦ The user must be authorized to **{get, update} "imagestream/layers"** in that project. The **system:image-pusher** role can be added to a user to provide these permissions. If you are a project administrator, then you would also have these permissions.

To create an image stream by manually pushing an image:

1. First, [log in to the internal registry](#).
2. Then, tag your image using the appropriate internal registry location. For example, if you had already pulled the **docker.io/centos:centos7** image locally:

```
$ docker tag docker.io/centos:centos7 172.30.48.125:5000/test/my-
image
```

3. Finally, push the image to your internal registry. For example:

```
$ docker push 172.30.48.125:5000/test/my-image
The push refers to a repository [172.30.48.125:5000/test/my-
image] (len: 1)
c8a648134623: Pushed
2bf4902415e3: Pushed
latest: digest:
sha256:be8bc4068b2f60cf274fc216e4caba6aa845fff5fa29139e6e7497bb57
e48d67 size: 6273
```

4. Verify that the image stream was created:

```
$ oc get is
NAME          DOCKER REPO          TAGS      UPDATED
my-image      172.30.48.125:5000/test/my-image  latest    3
seconds ago
```

CHAPTER 13. QUOTAS AND LIMIT RANGES

13.1. OVERVIEW

Using [quotas](#) and [limit ranges](#), cluster administrators can set constraints to limit the number of objects or amount of compute resources that are used in your project. This helps cluster administrators better manage and allocate resources across all projects, and ensure that no projects are using more than is appropriate for the cluster size.

As a developer, you can also set [requests and limits on compute resources](#) at the pod and container level.

The following sections help you understand how to check on your quota and limit range settings, what sorts of things they can constrain, and how you can request or limit compute resources in your own pods and containers.

13.2. QUOTAS

A resource quota, defined by a **ResourceQuota** object, provides constraints that limit aggregate resource consumption per project. It can limit the quantity of objects that can be created in a project by type, as well as the total amount of compute resources that may be consumed by resources in that project.



Note

Quotas are set by cluster administrators and are scoped to a given project.

13.2.1. Viewing Quotas

You can view usage statistics related to any hard limits defined in a project's quota by navigating in the web console to the project's **Settings** tab.

You can also use the CLI to view quota details:

1. First, get the list of quotas defined in the project. For example, for a project called **demoproject**:

```
$ oc get quota -n demoproject
NAME                AGE
besteffort          11m
compute-resources   2m
object-counts       29m
```

2. Then, describe the quota you are interested in, for example the **object-counts** quota:

```
$ oc describe quota object-counts -n demoproject
Name:      object-counts
Namespace: demoproject
Resource   Used Hard
-----

```



```

configmaps 3 10
persistentvolumeclaims 0 4
replicationcontrollers 3 20
secrets 9 10
services 2 10

```

Full quota definitions can be viewed by running **oc export** on the object. The following show some sample quota definitions:

Example 13.1. *object-counts.yaml*

```

apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
spec:
  hard:
    configmaps: "10" 1
    persistentvolumeclaims: "4" 2
    replicationcontrollers: "20" 3
    secrets: "10" 4
    services: "10" 5

```

1

The total number of **ConfigMap** objects that can exist in the project.

2

The total number of persistent volume claims (PVCs) that can exist in the project.

3

The total number of replication controllers that can exist in the project.

4

The total number of secrets that can exist in the project.

5

The total number of services that can exist in the project.

Example 13.2. *compute-resources.yaml*

```

apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4" 1
    requests.cpu: "1" 2
    requests.memory: 1Gi 3
    limits.cpu: "2" 4
    limits.memory: 2Gi 5

```

1

The total number of pods in a non-terminal state that can exist in the project.

2

Across all pods in a non-terminal state, the sum of CPU requests cannot exceed 1 core.

3

Across all pods in a non-terminal state, the sum of memory requests cannot exceed 1Gi.

4

Across all pods in a non-terminal state, the sum of CPU limits cannot exceed 2 cores.

5

Across all pods in a non-terminal state, the sum of memory limits cannot exceed 2Gi.

Example 13.3. *besteffort.yaml*

```

apiVersion: v1
kind: ResourceQuota
metadata:
  name: besteffort
spec:
  hard:
    pods: "1" 1
  scopes:
    - BestEffort 2

```

1

The total number of pods in a non-terminal state with **BestEffort** quality of service that can exist in the project.

2

Restricts the quota to only matching pods that have **BestEffort** quality of service for either memory or CPU.

Example 13.4. *compute-resources-long-running.yaml*

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-long-running
spec:
  hard:
    pods: "4" 1
    limits.cpu: "4" 2
    limits.memory: "2Gi" 3
  scopes:
    - NotTerminating 4
```

1

The total number of pods in a non-terminal state.

2

Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.

3

Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.

4

Restricts the quota to only matching pods where **spec.activeDeadlineSeconds** is **nil**. For example, this quota would not charge for build or deployer pods.

Example 13.5. *compute-resources-time-bound.yaml*

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-time-bound
spec:
  hard:
    pods: "2" 1
    limits.cpu: "1" 2
    limits.memory: "1Gi" 3
  scopes:
    - Terminating 4
```

1

The total number of pods in a non-terminal state.

2

Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.

3

Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.

4

Restricts the quota to only matching pods where **spec.activeDeadlineSeconds >=0**. For example, this quota would charge for build or deployer pods, but not long running pods like a web server or database.

13.2.2. Resources Managed by Quota

The following describes the set of compute resources and object types that may be managed by a quota.



Note

A pod is in a terminal state if **status.phase in (Failed, Succeeded)** is true.

Table 13.1. Compute Resources Managed by Quota

Resource Name	Description
cpu	The sum of CPU requests across all pods in a non-terminal state cannot exceed this value. cpu and requests.cpu are the same value and can be used interchangeably.
memory	The sum of memory requests across all pods in a non-terminal state cannot exceed this value. memory and requests.memory are the same value and can be used interchangeably.
requests.cpu	The sum of CPU requests across all pods in a non-terminal state cannot exceed this value. cpu and requests.cpu are the same value and can be used interchangeably.
requests.memory	The sum of memory requests across all pods in a non-terminal state cannot exceed this value. memory and requests.memory are the same value and can be used interchangeably.
requests.storage	The sum of storage requests across all persistent volume claims cannot exceed this value. storage and requests.storage are the same value and can be used interchangeably.
limits.cpu	The sum of CPU limits across all pods in a non-terminal state cannot exceed this value.
limits.memory	The sum of memory limits across all pods in a non-terminal state cannot exceed this value.
limits.storage	The sum of storage limits across all persistent volume claims cannot exceed this value.

Table 13.2. Object Counts Managed by Quota

Resource Name	Description
Pods	The total number of pods in a non-terminal state that can exist in the project.

Resource Name	Description
replicationcontrollers	The total number of replication controllers that can exist in the project.
resourcequotas	The total number of resource quotas that can exist in the project.
services	The total number of services that can exist in the project.
secrets	The total number of secrets that can exist in the project.
configmaps	The total number of ConfigMap objects that can exist in the project.
persistentvolumeclaims	The total number of persistent volume claims that can exist in the project.

13.2.3. Quota Scopes

Each quota can have an associated set of *scopes*. A quota will only measure usage for a resource if it matches the intersection of enumerated scopes.

Adding a scope to a quota restricts the set of resources to which that quota can apply. Specifying a resource outside of the allowed set results in a validation error.

Scope	Description
Terminating	Match pods where spec.activeDeadlineSeconds >= 0 .
NotTerminating	Match pods where spec.activeDeadlineSeconds is nil .
BestEffort	Match pods that have best effort quality of service for either cpu or memory .
NotBestEffort	Match pods that do not have best effort quality of service for cpu and memory .

A **BestEffort** scope restricts a quota to limiting the following resources:

- ✧ **pods**

A **Terminating**, **NotTerminating**, or **NotBestEffort** scope restricts a quota to tracking the following resources:

- ✧ **pods**

- ✧ **memory**

- ✧ **requests.memory**

- ✧ **limits.memory**

- ✧ **cpu**

- ✧ **requests.cpu**

- ✧ **limits.cpu**

13.2.4. Quota Enforcement

After a resource quota for a project is first created, the project restricts the ability to create any new resources that may violate a quota constraint until it has calculated updated usage statistics.

After a quota is created and usage statistics are updated, the project accepts the creation of new content. When you create or modify resources, your quota usage is incremented immediately upon the request to create or modify the resource.

When you delete a resource, your quota use is decremented during the next full recalculation of quota statistics for the project. If project modifications exceed a quota usage limit, the server denies the action. An appropriate error message is returned explaining the quota constraint violated, and what your currently observed usage stats are in the system.

13.2.5. Requests vs Limits

When allocating [compute resources](#), each container may specify a request and a limit value each for CPU and memory. Quotas can restrict any of these values.

If the quota has a value specified for **requests.cpu** or **requests.memory**, then it requires that every incoming container make an explicit request for those resources. If the quota has a value specified for **limits.cpu** or **limits.memory**, then it requires that every incoming container specify an explicit limit for those resources.

See [Compute Resources](#) for more on setting requests and limits in pods and containers.

13.3. LIMIT RANGES

A limit range, defined by a **LimitRange** object, enumerates [compute resource constraints](#) in a [project](#) at the pod and container level, and specifies the amount of resources that a pod or container can consume.

All resource create and modification requests are evaluated against each **LimitRange** object in the project. If the resource violates any of the enumerated constraints, then the resource is rejected. If the resource does not set an explicit value, and if the constraint supports a default value, then the default value is applied to the resource.

**Note**

Limit ranges are set by cluster administrators and are scoped to a given project.

13.3.1. Viewing Limit Ranges

You can view any limit ranges defined in a project by navigating in the web console to the project's **Settings** tab.

You can also use the CLI to view limit range details:

1. First, get the list of limit ranges defined in the project. For example, for a project called **demoproject**:

```
$ oc get limits -n demoproject
NAME              AGE
resource-limits   6d
```

2. Then, describe the limit range you are interested in, for example the **resource-limits** limit range:

```
<<<<<< HEAD
$ oc describe limits resource-limits
Name:      resource-limits
Namespace: demoproject
Type      Resource Min Max Default Request Default Limit Max
Limit/Request Ratio
-----
-----
Pod   cpu    30m 2 - - -
Pod   memory 150Mi 1Gi - - -
Container memory 150Mi 1Gi 307Mi 512Mi -
Container cpu    30m 2 60m 1 -
=====
$ oc describe limits resource-limits -n demoproject
Name:      resource-limits
Namespace: demoproject
Type      Resource Min
Max      Default Request Default Limit Max Limit/Request Ratio
-----
-----
-
Pod      cpu      200m
2        -
Pod      memory   6Mi
1Gi      -
Container cpu      100m
2        200m      300m      10
Container memory   4Mi
1Gi      100Mi      200Mi
openshift.io/Image storage -
1Gi      -
```



```

openshift.io/ImageStream      openshift.io/image      -
12      -      -      -
openshift.io/ImageStream      openshift.io/image-tags -
10      -      -      -
>>>>>> 7fb6456... Fix `oc describe limits` example

```

Full limit range definitions can be viewed by running **oc export** on the object. The following shows an example limit range definition:

Example 13.6. Limit Range Object Definition

```

apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" 1
spec:
  limits:
  -
    type: "Pod"
    max:
      cpu: "2" 2
      memory: "1Gi" 3
    min:
      cpu: "200m" 4
      memory: "6Mi" 5
  -
    type: "Container"
    max:
      cpu: "2" 6
      memory: "1Gi" 7
    min:
      cpu: "100m" 8
      memory: "4Mi" 9
    default:
      cpu: "300m" 10
      memory: "200Mi" 11
    defaultRequest:
      cpu: "200m" 12
      memory: "100Mi" 13
    maxLimitRequestRatio:
      cpu: "10" 14

```

1

The name of the limit range object.

2

The maximum amount of CPU that a pod can request on a node across all containers.

3

The maximum amount of memory that a pod can request on a node across all containers.

4

The minimum amount of CPU that a pod can request on a node across all containers.

5

The minimum amount of memory that a pod can request on a node across all containers.

6

The maximum amount of CPU that a single container in a pod can request.

7

The maximum amount of memory that a single container in a pod can request.

8

The minimum amount of CPU that a single container in a pod can request.

9

The minimum amount of memory that a single container in a pod can request.

10

The default amount of CPU that a container will be limited to use if not specified.

11

The default amount of memory that a container will be limited to use if not specified.

12

The default amount of CPU that a container will request to use if not specified.

13

The default amount of memory that a container will request to use if not specified.

14

The maximum amount of CPU burst that a container can make as a ratio of its limit over request.

13.3.2. Container Limits

Supported Resources:

- ✎ CPU
- ✎ Memory

Supported Constraints:

Per container, the following must hold true if specified:

Table 13.3. Container

Constraint	Behavior
Min	<p>Min[resource] less than or equal to container.resources.requests[resource] (required) less than or equal to container/resources.limits[resource] (optional)</p> <p>If the configuration defines a min CPU, then the request value must be greater than the CPU value. A limit value does not need to be specified.</p>
Max	<p>container.resources.limits[resource] (required) less than or equal to Max[resource]</p> <p>If the configuration defines a max CPU, then you do not need to define a request value, but a limit value does need to be set that satisfies the maximum CPU constraint.</p>

Constraint	Behavior
MaxLimitRequestRatio	<p>MaxLimitRequestRatio[resource] less than or equal to (container.resources.limits[resource] / container.resources.requests[resource])</p> <p>If a configuration defines a maxLimitRequestRatio value, then any new containers must have both a request and limit value. Additionally, OpenShift Enterprise calculates a limit to request ratio by dividing the limit by the request.</p> <p>For example, if a container has cpu: 500 in the limit value, and cpu: 100 in the request value, then its limit to request ratio for cpu is 5. This ratio must be less than or equal to the maxLimitRequestRatio.</p>

Supported Defaults:**Default[resource]**

Defaults **container.resources.limit[resource]** to specified value if none.

Default Requests[resource]

Defaults **container.resources.requests[resource]** to specified value if none.

13.3.3. Pod Limits**Supported Resources:**

 CPU

 Memory

Supported Constraints:

Across all containers in a pod, the following must hold true:

Table 13.4. Pod

Constraint	Enforced Behavior
Min	<p>Min[resource] less than or equal to container.resources.requests[resource] (required) less than or equal to container.resources.limits[resource] (optional)</p>
Max	<p>container.resources.limits[resource] (required) less than or equal to Max[resource]</p>

Constraint	Enforced Behavior
MaxLimitRequestRatio	MaxLimitRequestRatio[resource] less than or equal to (container.resources.limits[resource] / container.resources.requests[resource])

13.4. COMPUTE RESOURCES

Each container running on a node consumes compute resources, which are measurable quantities that can be requested, allocated, and consumed.

When authoring a pod configuration file, you can optionally specify how much CPU and memory (RAM) each container needs in order to better schedule pods in the cluster and ensure satisfactory performance.

CPU is measured in units called millicores. Each node in a cluster inspects the operating system to determine the amount of CPU cores on the node, then multiplies that value by 1000 to express its total capacity. For example, if a node has 2 cores, the node's CPU capacity would be represented as 2000m. If you wanted to use 1/10 of a single core, it would be represented as 100m.

Memory is measured in bytes. In addition, it may be used with SI suffices (E, P, T, G, M, K) or their power-of-two-equivalents (Ei, Pi, Ti, Gi, Mi, Ki).

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - image: nginx
    name: nginx
    resources:
      requests:
        cpu: 100m 1
        memory: 200Mi 2
      limits:
        cpu: 200m 3
        memory: 400Mi 4
```

1

The container requests 100m cpu.

2

The container requests 200Mi memory.

3

— The container requests 200m

The container limits 200m cpu.

4

The container limits 400Mi memory.

13.4.1. CPU Requests

Each container in a pod can specify the amount of CPU it requests on a node. The scheduler uses CPU requests to find a node with an appropriate fit for a container.

The CPU request represents a minimum amount of CPU that your container may consume, but if there is no contention for CPU, it can use all available CPU on the node. If there is CPU contention on the node, CPU requests provide a relative weight across all containers on the system for how much CPU time the container may use.

On the node, CPU requests map to Kernel CFS shares to enforce this behavior.

13.4.2. Viewing Compute Resources

To view compute resources for a pod:

```
$ oc describe pod nginx-tfjxt
Name:          nginx-tfjxt
Namespace:     default
Image(s):      nginx
Node:          /
Labels:        run=nginx
Status:        Pending
Reason:
Message:
IP:
Replication Controllers:  nginx (1/1 replicas created)
Containers:
  nginx:
    Container ID:
    Image:        nginx
    Image ID:
    QoS Tier:
      cpu:        Burstable
      memory:     Burstable
    Limits:
      cpu:        200m
      memory:     400Mi
    Requests:
      cpu:        100m
      memory:     200Mi
    State:        Waiting
    Ready:        False
    Restart Count: 0
    Environment Variables:
```

13.4.3. CPU Limits

Each container in a pod can specify the amount of CPU it is limited to use on a node. CPU limits control the maximum amount of CPU that your container may use independent of contention on the node. If a container attempts to exceed the specified limit, the system will throttle the container. This allows the container to have a consistent level of service independent of the number of pods scheduled to the node.

13.4.4. Memory Requests

By default, a container is able to consume as much memory on the node as possible. In order to improve placement of pods in the cluster, specify the amount of memory required for a container to run. The scheduler will then take available node memory capacity into account prior to binding your pod to a node. A container is still able to consume as much memory on the node as possible even when specifying a request.

13.4.5. Memory Limits

If you specify a memory limit, you can constrain the amount of memory the container can use. For example, if you specify a limit of 200Mi, a container will be limited to using that amount of memory on the node. If the container exceeds the specified memory limit, it will be terminated and potentially restarted dependent upon the container restart policy.

13.4.6. Quality of Service Tiers

A compute resource is classified with a *quality of service* (QoS) based on the specified request and limit value.

Quality of Service	Description
BestEffort	Provided when a request and limit are not specified.
Burstable	Provided when a request is specified that is less than an optionally specified limit.
Guaranteed	Provided when a limit is specified that is equal to an optionally specified request.

A container may have a different quality of service for each compute resource. For example, a container can have **Burstable** CPU and **Guaranteed** memory qualities of service.

The quality of service has different impacts on different resources, depending on whether the resource is compressible or not. CPU is a compressible resource, whereas memory is an incompressible resource.

With CPU Resources:

- ✦ A **BestEffort CPU** container is able to consume as much CPU as is available on a node but runs with the lowest priority.

- ✦ A **Burstable CPU** container is guaranteed to get the minimum amount of CPU requested, but it may or may not get additional CPU time. Excess CPU resources are distributed based on the amount requested across all containers on the node.
- ✦ A **Guaranteed CPU** container is guaranteed to get the amount requested and no more, even if there are additional CPU cycles available. This provides a consistent level of performance independent of other activity on the node.

With Memory Resources:

- ✦ A **BestEffort memory** container is able to consume as much memory as is available on the node, but there are no guarantees that the scheduler will place that container on a node with enough memory to meet its needs. In addition, a **BestEffort** container has the greatest chance of being killed if there is an out of memory event on the node.
- ✦ A **Burstable memory** container is scheduled on the node to get the amount of memory requested, but it may consume more. If there is an out of memory event on the node, **Burstable** containers are killed after **BestEffort** containers when attempting to recover memory.
- ✦ A **Guaranteed memory** container gets the amount of memory requested, but no more. In the event of an out of memory event, it will only be killed if there are no more **BestEffort** or **Burstable** containers on the system.

13.4.7. Specifying Compute Resources via CLI

To specify compute resources via the CLI:

```
$ oc run nginx --image=nginx --limits=cpu=200m,memory=400Mi --  
requests=cpu=100m,memory=200Mi
```

13.5. PROJECT RESOURCE LIMITS

[Resource limits can be set per-project](#) by cluster administrators. Developers do not have the ability to create, edit, or delete these limits, but can [view them](#) for projects they have access to.

CHAPTER 14. DEPLOYMENTS

14.1. OVERVIEW

A deployment in OpenShift Enterprise is a replication controller based on a user defined template called a deployment configuration. Deployments are created manually or in response to triggered events.

The deployment system provides:

- ✧ A [deployment configuration](#), which is a template for deployments.
- ✧ [Triggers](#) that drive automated deployments in response to events.
- ✧ User-customizable [strategies](#) to transition from the previous deployment to the new deployment.
- ✧ [Rollbacks](#) to a previous deployment.
- ✧ Manual replication [scaling](#).

The deployment configuration contains a version number that is incremented each time a new deployment is created from that configuration. In addition, the cause of the last deployment is added to the configuration.

14.2. CREATING A DEPLOYMENT CONFIGURATION

A deployment configuration consists of the following key parts:

- ✧ A replication controller template which describes the application to be deployed.
- ✧ The default replica count for the deployment.
- ✧ A deployment [strategy](#) which will be used to execute the deployment.
- ✧ A set of [triggers](#) which cause deployments to be created automatically.

Deployment configurations are **deploymentConfig** OpenShift Enterprise API resources which can be managed with the **oc** command like any other resource. The following is an example of a **deploymentConfig** resource:

```
kind: "DeploymentConfig"
apiVersion: "v1"
metadata:
  name: "frontend"
spec:
  template: 1
    metadata:
      labels:
        name: "frontend"
    spec:
      containers:
        - name: "helloworld"
          image: "openshift/origin-ruby-sample"
          ports:
            - containerPort: 8080
```

```

        protocol: "TCP"
replicas: 5 2
selector:
  name: "frontend"
triggers:
  - type: "ConfigChange" 3
  - type: "ImageChange" 4
    imageChangeParams:
      automatic: true
      containerNames:
        - "helloworld"
      from:
        kind: "ImageStreamTag"
        name: "origin-ruby-sample:latest"
strategy: 5
  type: "Rolling"

```

1

The replication controller template named **frontend** describes a simple Ruby application.

2

There will be 5 replicas of **frontend** by default.

3

A [configuration change trigger](#) causes a new deployment to be created any time the replication controller template changes.

4

An [image change trigger](#) trigger causes a new deployment to be created each time a new version of the **origin-ruby-sample:latest** image repository is available.

5

The [Rolling strategy](#) is the default and may be omitted.

14.3. STARTING A DEPLOYMENT

You can start a new deployment manually using the web console, or from the CLI:

```
$ oc deploy <deployment_config> --latest
```

**Note**

If there's already a deployment in progress, the command will display a message and a new deployment will not be started.

14.4. VIEWING A DEPLOYMENT

To get basic information about recent deployments:

```
$ oc deploy <deployment_config>
```

This will show details about the latest and recent deployments, including any currently running deployment.

For more detailed information about a deployment configuration and the latest deployment:

```
$ oc describe dc <deployment_config>
```

**Note**

The [web console](#) shows deployments in the **Browse** tab.

14.5. CANCELING A DEPLOYMENT

To cancel a running or stuck deployment:

```
$ oc deploy <deployment_config> --cancel
```

Warning

The cancellation is a best-effort operation, and may take some time to complete. It's possible the deployment will partially or totally complete before the cancellation is effective.

14.6. RETRYING A DEPLOYMENT

To retry the last failed deployment:

```
$ oc deploy <deployment_config> --retry
```

If the last deployment didn't fail, the command will display a message and the deployment will not be retried.

**Note**

Retrying a deployment restarts the deployment and does not create a new deployment version. The restarted deployment will have the same configuration it had when it failed.

14.7. ROLLING BACK A DEPLOYMENT

Rollbacks revert an application back to a previous deployment and can be performed using the REST API, the CLI, or the web console.

To rollback to the last successful deployment:

```
$ oc rollback <deployment_config>
```

The deployment configuration's template will be reverted to match the deployment specified in the rollback command, and a new deployment will be started.

Image change triggers on the deployment configuration are disabled as part of the rollback to prevent unwanted deployments soon after the rollback is complete. To re-enable the image change triggers:

```
$ oc deploy <deployment_config> --enable-triggers
```

To roll back to a specific version:

```
$ oc rollback <deployment_config> --to-version=1
```

To see what the rollback would look like without performing the rollback:

```
$ oc rollback <deployment_config> --dry-run
```

14.8. EXECUTING COMMANDS INSIDE A CONTAINER

You can add a command to a container, which modifies the container's startup behavior by overruling the image's **ENTRYPOINT**. This is different from a [lifecycle hook](#), which instead can be run once per deployment at a specified time.

Add the **command** parameters to the **spec** field of the deployment configuration. You can also add an **args** field, which modifies the **command** (or the **ENTRYPOINT** if **command** does not exist).

```
...
spec:
  containers:
  -
    name: <container_name>
    image: 'image'
    command:
    - '<command>'
    args:
```

```

- '<argument_1>'
- '<argument_2>'
- '<argument_3>'
...

```

For example, to execute the **java** command with the `-jar` and `/opt/app-root/springboots2idemo.jar` arguments:

```

...
spec:
  containers:
    -
      name: example-spring-boot
      image: 'image'
      command:
        - java
      args:
        - '-jar'
        - /opt/app-root/springboots2idemo.jar
...

```

14.9. VIEWING DEPLOYMENT LOGS

To view the logs of the latest deployment for a given deployment configuration:

```
$ oc logs dc/<deployment_config> [--follow]
```

Logs can be retrieved either while the deployment is running or if it has failed. If the deployment was successful, there will be no logs to view.

You can also view logs from older deployments:

```
$ oc logs --version=1 dc/<deployment_config>
```

This command returns the logs from the first deployment of the provided deployment configuration, if and only if that deployment exists (i.e., it has failed and has not been manually deleted or pruned).

14.10. TRIGGERS

A deployment configuration can contain triggers, which drive the creation of new deployments in response to events, only inside OpenShift Enterprise at the moment.

Warning

If no triggers are defined on a deployment configuration, deployments must be [started manually](#).

14.10.1. Configuration Change Trigger

The **ConfigChange** trigger results in a new deployment whenever new changes are detected in the pod template of the deployment configuration.



Note

If only a **ConfigChange** trigger is defined on a deployment configuration, the first deployment is automatically created soon after the deployment configuration itself is created.

Example 14.1. A ConfigChange Trigger

```
triggers:
  - type: "ConfigChange"
```

14.10.2. Image Change Trigger

The **ImageChange** trigger results in a new deployment whenever the value of an image stream tag changes, either by a build or because it was imported.

Example 14.2. An ImageChange Trigger

```
triggers:
  - type: "ImageChange"
    imageChangeParams:
      automatic: true 1
      from:
        kind: "ImageStreamTag"
        name: "origin-ruby-sample:latest"
      containerNames:
        - "helloworld"
```

1

If the `imageChangeParams.automatic` field is set to **false**, the trigger is disabled.

With the above example, when the **latest** tag value of the **origin-ruby-sample** image stream changes and the new image value differs from the current image specified in the deployment configuration's **helloworld** container, a new deployment is created using the new image for the **helloworld** container.



Note

If an **ImageChange** trigger is defined on a deployment configuration (with a **ConfigChange** trigger or with **automatic=true**) and the **ImageStreamTag** pointed by the **ImageChange** trigger does not exist yet, then the first deployment automatically starts as soon as an image is imported or pushed by a build to the **ImageStreamTag**.

14.11. STRATEGIES

A deployment strategy determines the deployment process, and is defined by the deployment configuration. Each application has different requirements for availability (and other considerations) during deployments. OpenShift Enterprise provides strategies to support a variety of deployment scenarios.

A deployment strategy uses [readiness checks](#) to determine if a new pod is ready for use. If a readiness check fails, the deployment is stopped.

The [Rolling strategy](#) is the default strategy used if no strategy is specified on a deployment configuration.

14.11.1. Rolling Strategy

The rolling strategy performs a rolling update and supports [lifecycle hooks](#) for injecting code into the deployment process.

The rolling deployment strategy waits for pods to pass their [readiness check](#) before scaling down old components, and does not allow pods that do not pass their readiness check within a configurable timeout.

The following is an example of the Rolling strategy:

```
strategy:
  type: Rolling
  rollingParams:
    timeoutSeconds: 120 1
    maxSurge: "20%" 2
    maxUnavailable: "10%" 3
    pre: {} 4
    post: {}
```

1

How long to wait for a scaling event before giving up. Optional; the default is 120.

2

maxSurge is optional and defaults to **25%**; see below.

3

maxUnavailable is optional and defaults to **25%**; see below.

4

pre and **post** are both [lifecycle hooks](#).

The Rolling strategy will:

1. Execute any **pre** lifecycle hook.
2. Scale up the new deployment based on the surge configuration.
3. Scale down the old deployment based on the max unavailable configuration.
4. Repeat this scaling until the new deployment has reached the desired replica count and the old deployment has been scaled to zero.
5. Execute any **post** lifecycle hook.

Important

When scaling down, the Rolling strategy waits for pods to become ready so it can decide whether further scaling would affect availability. If scaled up pods never become ready, the deployment will eventually time out and result in a deployment failure.

Important

When executing the **post** lifecycle hook, all failures will be ignored regardless of the failure policy specified on the hook.

The **maxUnavailable** parameter is the maximum number of pods that can be unavailable during the update. The **maxSurge** parameter is the maximum number of pods that can be scheduled above the original number of pods. Both parameters can be set to either a percentage (e.g., **10%**) or an absolute value (e.g., **2**). The default value for both is **25%**.

These parameters allow the deployment to be tuned for availability and speed. For example:

- ✳ **maxUnavailable=0** and **maxSurge=20%** ensures full capacity is maintained during the update and rapid scale up.
- ✳ **maxUnavailable=10%** and **maxSurge=0** performs an update using no extra capacity (an in-place update).
- ✳ **maxUnavailable=10%** and **maxSurge=10%** scales up and down quickly with some potential for capacity loss.

14.11.2. Recreate Strategy

The Recreate strategy has basic rollout behavior and supports [lifecycle hooks](#) for injecting code into the deployment process.

The following is an example of the Recreate strategy:

```
strategy:
  type: Recreate
  recreateParams: 1
    pre: {} 2
    mid: {}
    post: {}
```

1

recreateParams are optional.

2

pre, **mid**, and **post** are both [lifecycle hooks](#).

The Recreate strategy will:

1. Execute any "pre" lifecycle hook.
2. Scale down the previous deployment to zero.
3. Execute any "mid" lifecycle hook.
4. Scale up the new deployment.
5. Execute any "post" lifecycle hook.



Important

During scale up, if the replica count of the deployment is greater than one, the first replica of the deployment will be validated for readiness before fully scaling up the deployment. If the validation of the first replica fails, the deployment will be considered a failure.

14.11.3. Custom Strategy

The Custom strategy allows you to provide your own deployment behavior.

The following is an example of the Custom strategy:

```
strategy:
  type: Custom
  customParams:
    image: organization/strategy
    command: [ "command", "arg1" ]
    environment:
```

```
- name: ENV_1
  value: VALUE_1
```

In the above example, the **organization/strategy** container image provides the deployment behavior. The optional **command** array overrides any **CMD** directive specified in the image's **Dockerfile**. The optional environment variables provided are added to the execution environment of the strategy process.

Additionally, OpenShift Enterprise provides the following environment variables to the strategy process:

Environment Variable	Description
OPENSSHIFT_DEPLOYMENT_NAME	The name of the new deployment (a replication controller).
OPENSSHIFT_DEPLOYMENT_NAMESPACE	The namespace of the new deployment.

The replica count of the new deployment will initially be zero. The responsibility of the strategy is to make the new deployment active using the logic that best serves the needs of the user.

14.12. LIFECYCLE HOOKS

The [Recreate](#) and [Rolling](#) strategies support lifecycle hooks, which allow behavior to be injected into the deployment process at predefined points within the strategy:

The following is an example of a "pre" lifecycle hook:

```
pre:
  failurePolicy: Abort
  execNewPod: {} 1
```

1

execNewPod is [a pod-based lifecycle hook](#).

Every hook has a **failurePolicy**, which defines the action the strategy should take when a hook failure is encountered:

Abort	The deployment should be considered a failure if the hook fails.
Retry	The hook execution should be retried until it succeeds.

Ignore

Any hook failure should be ignored and the deployment should proceed.

Warning

Some hook points for a strategy might support only a subset of failure policy values. For example, the [Recreate](#) and [Rolling](#) strategies do not currently support the **Abort** policy for a "post" deployment lifecycle hook. Consult the documentation for a given strategy for details on any restrictions regarding lifecycle hooks.

Hooks have a type-specific field that describes how to execute the hook. Currently [pod-based hooks](#) are the only supported hook type, specified by the **execNewPod** field.

14.12.1. Pod-based Lifecycle Hook

Pod-based lifecycle hooks execute hook code in a new pod derived from the template in a deployment configuration.

The following simplified example deployment configuration uses the [Rolling strategy](#). Triggers and some other minor details are omitted for brevity:

```
kind: DeploymentConfig
apiVersion: v1
metadata:
  name: frontend
spec:
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - name: helloworld
          image: openshift/origin-ruby-sample
  replicas: 5
  selector:
    name: frontend
  strategy:
    type: Rolling
    rollingParams:
      pre:
        failurePolicy: Abort
        execNewPod:
          containerName: helloworld 1
          command: [ "/usr/bin/command", "arg1", "arg2" ] 2
          env: 3
```

```

      - name: CUSTOM_VAR1
        value: custom_value1
    volumes:
      - data 4

```

1

The **helloworld** name refers to `spec.template.spec.containers[0].name`.

2

This **command** overrides any **ENTRYPOINT** defined by the **openshift/origin-ruby-sample** image.

3

env is an optional set of environment variables for the hook container.

4

volumes is an optional set of volume references for the hook container.

In this example, the "pre" hook will be executed in a new pod using the **openshift/origin-ruby-sample** image from the **helloworld** container. The hook pod will have the following properties:

- ✎ The hook command will be `/usr/bin/command arg1 arg2`.
- ✎ The hook container will have the **CUSTOM_VAR1=custom_value1** environment variable.
- ✎ The hook failure policy is **Abort**, meaning the deployment will fail if the hook fails.
- ✎ The hook pod will inherit the **data** volume from the deployment configuration pod.

14.13. DEPLOYMENT RESOURCES

A deployment is completed by a pod that consumes resources (memory and CPU) on a node. By default, pods consume unbounded node resources. However, if a project specifies default container limits, then pods consume resources up to those limits.

You can also limit resource use by specifying resource limits as part of the deployment strategy. Deployment resources can be used with the Recreate, Rolling, or Custom deployment strategies.

In the following example, each of **resources**, **cpu**, and **memory** is optional:

```

type: "Recreate"
resources:
  limits:
    cpu: "100m" 1
    memory: "256Mi" 2

```

1

cpu is in CPU units: **100m** represents 0.1 CPU units ($100 * 1e-3$).

2

memory is in bytes: **256Mi** represents 268435456 bytes ($256 * 2^{20}$).

However, if a quota has been defined for your project, one of the following two items is required:

- ✦ A **resources** section set with an explicit **requests**:

```
type: "Recreate"
resources:
  requests: 1
    cpu: "100m"
    memory: "256Mi"
```

1

The **requests** object contains the list of resources that correspond to the list of resources in the quota.

- ✦ A [limit range](#) defined in your project, where the defaults from the **LimitRange** object apply to pods created during the deployment process.

Otherwise, deploy pod creation will fail, citing a failure to satisfy quota.

14.14. MANUAL SCALING

In addition to rollbacks, you can exercise fine-grained control over the number of replicas from the web console, or by using the **oc scale** command. For example, the following command sets the replicas in the deployment configuration **frontend** to 3.

```
$ oc scale dc frontend --replicas=3
```

The number of replicas eventually propagates to the desired and current state of the deployment configured by the deployment configuration **frontend**.

14.15. ASSIGNING PODS TO SPECIFIC NODES

You can use node selectors in conjunction with labeled nodes to control pod placement.



Note

OpenShift Enterprise administrators can assign labels [during an advanced installation](#), or [added to a node after installation](#).

Cluster administrators [can set the default node selector](#) for your project in order to restrict pod placement to specific nodes. As an OpenShift developer, you can set a node selector on a pod configuration to restrict nodes even further.

To add a node selector when creating a pod, edit the pod configuration, and add the **nodeSelector** value. This can be added to a single pod configuration, or in a pod template:

```
apiVersion: v1
kind: Pod
spec:
  nodeSelector:
    disktype: ssd
  ...
```

Pods created when the node selector is in place are assigned to nodes with the specified labels.

The labels specified here are used in conjunction with the labels [added by a cluster administrator](#). For example, if a project has the **type=user-node** and **region=east** labels added to a project by the cluster administrator, and you add the above **disktype: ssd** label to a pod, the pod will only ever be scheduled on nodes that have all three labels.



Note

Labels can only be set to one value, so setting a node selector of **region=west** in a pod configuration that has **region=east** as the administrator-set default, results in a pod that will never be scheduled.

14.16. RUNNING A POD WITH A DIFFERENT SERVICE ACCOUNT

You can run a pod with a service account other than the default:

1. Edit the deployment configuration:

```
$ oc edit dc/<deployment_config>
```

2. Add the **serviceAccount** and **serviceAccountName** parameters to the **spec** field, and specify the service account you want to use:

```
spec:
  securityContext: {}
  serviceAccount: <service_account>
  serviceAccountName: <service_account>
```

CHAPTER 15. ROUTES

15.1. OVERVIEW

An OpenShift Enterprise [route](#) exposes a [service](#) at a host name, like *www.example.com*, so that external clients can reach it by name.

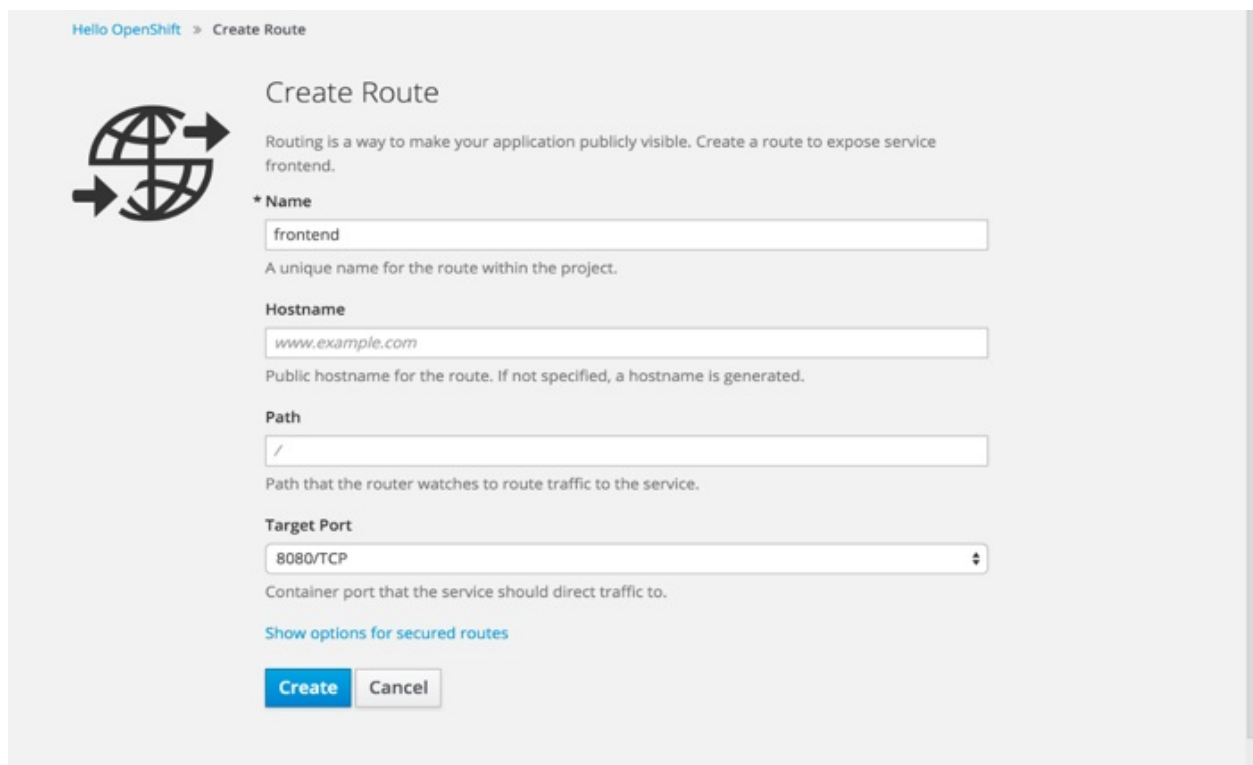
DNS resolution for a host name is handled separately from routing; your administrator may have configured a cloud domain that will always correctly resolve to the OpenShift Enterprise router, or if using an unrelated host name you may need to modify its DNS records independently to resolve to the router.

15.2. CREATING ROUTES

You can create unsecured and secured routes using the web console or the CLI.

Using the web console, you can navigate to the **Browse** → **Routes** page, then click **Create Route** to define and create a route in your project:

Figure 15.1. Creating a Route Using the Web Console



The screenshot shows the 'Create Route' form in the OpenShift web console. The breadcrumb navigation at the top reads 'Hello OpenShift > Create Route'. On the left is a globe icon with two arrows. The main heading is 'Create Route'. Below it is a descriptive text: 'Routing is a way to make your application publicly visible. Create a route to expose service frontend.' The form contains the following fields:

- * Name:** A text input field containing 'frontend'. Below it is a hint: 'A unique name for the route within the project.'
- Hostname:** A text input field containing 'www.example.com'. Below it is a hint: 'Public hostname for the route. If not specified, a hostname is generated.'
- Path:** A text input field containing '/'. Below it is a hint: 'Path that the router watches to route traffic to the service.'
- Target Port:** A dropdown menu showing '8080/TCP'. Below it is a hint: 'Container port that the service should direct traffic to.'

Below the fields is a link that says 'Show options for secured routes'. At the bottom are two buttons: 'Create' (in blue) and 'Cancel' (in gray).

Using the CLI, the following example creates an unsecured route:

```
$ oc expose svc/frontend --hostname=www.example.com
```

The new route inherits the name from the service unless you specify one using the **--name** option.

Example 15.1. YAML Definition of the Unsecured Route Created Above

```

apiVersion: v1
kind: Route
metadata:
  name: frontend
spec:
  host: www.example.com
  to:
    kind: Service
    name: frontend

```

Unsecured routes are the default configuration, and are therefore the simplest to set up. However, [secured routes](#) offer security for connections to remain private. To create a secured HTTPS route encrypted with a key and certificate (PEM-format files which you must generate and sign separately), you can use the **create route** command and optionally provide certificates and a key.



Note

[TLS](#) is the replacement of SSL for HTTPS and other encrypted protocols.

```

$ oc create route edge --service=frontend \
  --cert=${MASTER_CONFIG_DIR}/ca.crt \
  --key=${MASTER_CONFIG_DIR}/ca.key \
  --ca-cert=${MASTER_CONFIG_DIR}/ca.crt \
  --hostname=www.example.com

```

Example 15.2. YAML Definition of the Secured Route Created Above

```

apiVersion: v1
kind: Route
metadata:
  name: frontend
spec:
  host: www.example.com
  to:
    kind: Service
    name: frontend
  tls:
    termination: edge
    key: |-
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----

```



```
caCertificate: |-
-----BEGIN CERTIFICATE-----
[...]
-----END CERTIFICATE-----
```

Currently, password protected key files are not supported. HAProxy prompts for a password upon starting and does not have a way to automate this process. To remove a passphrase from a keyfile, you can run:

```
# openssl rsa -in <passwordProtectedKey.key> -out <new.key>
```

You can create a secured route without specifying a key and certificate, in which case the [router's default certificate](#) will be used for TLS termination.



Note

TLS termination in OpenShift Enterprise relies on [SNI](#) for serving custom certificates. Any non-SNI traffic received on port 443 is handled with TLS termination and a default certificate, which may not match the requested host name, resulting in validation errors.

Further information on all types of [TLS termination](#) as well as [path-based routing](#) are available in the [Architecture section](#).

CHAPTER 16. INTEGRATING EXTERNAL SERVICES

16.1. OVERVIEW

Many OpenShift Enterprise applications use external resources, such as external databases, or an external SaaS endpoint. These external resources can be modeled as native OpenShift Enterprise services, so that applications can work with them as they would any other internal service.

16.2. EXTERNAL MYSQL DATABASE

One of the most common types of external services is an external database. To support an external database, an application needs:

1. An endpoint to communicate with.
2. A set of credentials and coordinates, including:
 - a. A user name
 - b. A passphrase
 - c. A database name

The solution for integrating with an external database includes:

- ✳ A **Service** object to represent the SaaS provider as an OpenShift Enterprise service.
- ✳ One or more **Endpoints** for the service.
- ✳ Environment variables in the appropriate pods containing the credentials.

The following steps outline a scenario for integrating with an external MySQL database:

1. Create an [OpenShift Enterprise service](#) to represent your external database. This is similar to creating an internal service; the difference is in the service's **Selector** field.

Internal OpenShift Enterprise services use the **Selector** field to associate pods with services using [labels](#). The **EndpointsController** system component synchronizes the endpoints for services that specify selectors with the pods that match the selector. The [service proxy](#) and OpenShift Enterprise [router](#) load-balance requests to the service amongst the service's endpoints.

Services that represent an external resource do not require associated pods. Instead, leave the **Selector** field unset. This represents the external service, making the **EndpointsController** ignore the service and allows you to specify endpoints manually:

```
kind: "Service"
apiVersion: "v1"
metadata:
  name: "external-mysql-service"
spec:
  ports:
    -
```

```

    name: "mysql"
    protocol: "TCP"
    port: 3306
    targetPort: 3306
    nodePort: 0
  selector: {} 1

```

1

The **selector** field to leave blank.

2. Next, create the required endpoints for the service. This gives the service proxy and router the location to send traffic directed to the service:

```

kind: "Endpoints"
apiVersion: "v1"
metadata:
  name: "external-mysql-service" 1
subsets: 2
-
  addresses:
  -
    ip: "10.0.0.0" 3
  ports:
  -
    port: 3306 4
    name: "mysql"

```

1

The name of the **Service** instance, as defined in the previous step.

2

Traffic to the service will be load-balanced between the supplied **Endpoints** if more than one is supplied.

3

Endpoints IPs **cannot be** loopback (127.0.0.0/8), link-local (169.254.0.0/16), or link-local multicast (224.0.0.0/24).

4

The **port** and **name** definition must match the **port** and **name** value in the service defined in the previous step.

- Now that the service and endpoints are defined, give the appropriate pods access to the credentials to use the service by setting environment variables in the appropriate containers:

```
kind: "DeploymentConfig"
apiVersion: "v1"
metadata:
  name: "my-app-deployment"
spec: ❶
  strategy:
    type: "Rolling"
    rollingParams:
      updatePeriodSeconds: 1
      intervalSeconds: 1
      timeoutSeconds: 120
  replicas: 2
  selector:
    name: "frontend"
  template:
    metadata:
      labels:
        name: "frontend"
    spec:
      containers:
        -
          name: "helloworld"
          image: "origin-ruby-sample"
          ports:
            -
              containerPort: 3306
              protocol: "TCP"
          env:
            -
              name: "MYSQL_USER"
              value: "${MYSQL_USER}" ❷
            -
              name: "MYSQL_PASSWORD"
              value: "${MYSQL_PASSWORD}" ❸
            -
              name: "MYSQL_DATABASE"
              value: "${MYSQL_DATABASE}" ❹
```

1

Other fields on the **DeploymentConfig** are omitted

2

The user name to use with the service.

3

The passphrase to use with the service.

4

The database name.

External Database Environment Variables

Using an external service in your application is similar to using an internal service. Your application will be assigned environment variables for the service and the additional environment variables with the credentials described in the previous step. For example, a MySQL container receives the following environment variables:

- ✎ `EXTERNAL_MYSQL_SERVICE_SERVICE_HOST=<ip_address>`
- ✎ `EXTERNAL_MYSQL_SERVICE_SERVICE_PORT=<port_number>`
- ✎ `MYSQL_USERNAME=<mysql_username>`
- ✎ `MYSQL_PASSPHRASE=<mysql_passphrase>`
- ✎ `MYSQL_DATABASE_NAME=<mysql_database>`

The application is responsible for reading the coordinates and credentials for the service from the environment and establishing a connection with the database via the service.

16.3. EXTERNAL SAAS PROVIDER

A common type of external service is an external SaaS endpoint. To support an external SaaS provider, an application needs:

1. An endpoint to communicate with
2. A set of credentials, such as:
 - a. An API key
 - b. A user name
 - c. A passphrase

The following steps outline a scenario for integrating with an external SaaS provider:

1. Create an [OpenShift Enterprise service](#) to represent the external service. This is similar to creating an internal service; however the difference is in the service's **Selector** field.

Internal OpenShift Enterprise services use the **Selector** field to associate pods with services using [labels](#). A system component called **EndpointsController** synchronizes the endpoints for services that specify selectors with the pods that match the selector. The [service proxy](#) and OpenShift Enterprise [router](#) load-balance requests to the service amongst the service's endpoints.

Services that represents an external resource do not require that pods be associated with it. Instead, leave the **Selector** field unset. This makes the **EndpointsController** ignore the service and allows you to specify endpoints manually:

```
kind: "Service"
apiVersion: "v1"
metadata:
  name: "example-external-service"
spec:
  ports:
  -
    name: "mysql"
    protocol: "TCP"
    port: 3306
    targetPort: 3306
    nodePort: 0
  selector: {} 1
```

1

The **selector** field to leave blank.

2. Next, create endpoints for the service containing the information about where to send traffic directed to the service proxy and the router:

```
kind: "Endpoints"
apiVersion: "v1"
metadata:
  name: "example-external-service" 1
subsets: 2
- addresses:
  - ip: "10.10.1.1"
  ports:
  - name: "mysql"
    port: 3306
```

1

The name of the **Service** instance.

Traffic to the service is load-balanced between the **subsets** supplied here.

1. Now that the service and endpoints are defined, give pods the credentials to use the service by setting environment variables in the appropriate containers:

```
---
```

```

kind: "DeploymentConfig"
apiVersion: "v1"
metadata:
  name: "my-app-deployment"
spec: ❶
  strategy:
    type: "Rolling"
    rollingParams:
      updatePeriodSeconds: 1
      intervalSeconds: 1
      timeoutSeconds: 120
  replicas: 1
  selector:
    name: "frontend"
  template:
    metadata:
      labels:
        name: "frontend"
    spec:
      containers:
        -
          name: "helloworld"
          image: "openshift/openshift/origin-ruby-
sample"
          ports:
            -
              containerPort: 3306
              protocol: "TCP"
          env:
            -
              name: "SAAS_API_KEY" ❷
              value: "<SaaS service API key>"
            -
              name: "SAAS_USERNAME" ❸
              value: "<SaaS service user>"
            -
              name: "SAAS_PASSPHRASE" ❹
              value: "<SaaS service passphrase>"

```

1

1

Other fields on the **DeploymentConfig** are omitted.

2

2

SAAS_API_KEY: The API key to use with the service.

3

SAAS_USERNAME: The user name to use with the service.

SAAS_PASSPHRASE: The passphrase to use with the service.

External SaaS Provider Environment Variables

Similarly, when using an internal service, your application is assigned environment variables for the service and the additional environment variables with the credentials described in the above steps. In the above example, the container receives the following environment variables:

✿ **EXAMPLE_EXTERNAL_SERVICE_SERVICE_HOST=<ip_address>**

✿ **EXAMPLE_EXTERNAL_SERVICE_SERVICE_PORT=<port_number>**

✿ **SAAS_API_KEY=<saas_api_key>**

✿ **SAAS_USERNAME=<saas_username>**

✿ **SAAS_PASSPHRASE=<saas_passphrase>**

The application reads the coordinates and credentials for the service from the environment and establishes a connection with the service.

CHAPTER 17. SECRETS

17.1. OVERVIEW

The **Secret** object type provides a mechanism to hold sensitive information such as passwords, OpenShift Enterprise client config files, **dockercfg** files, private source repository credentials, etc. Secrets decouple sensitive content from the pods that use it and can be mounted into containers using a volume plug-in or used by the system to perform actions on behalf of a pod. This topic discusses important properties of secrets and provides an overview on how developers can use them.

Example 17.1. YAML Secret Object Definition

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
  namespace: my-namespace
data: 1
  username: "dmFsdWUtMQ0K"
  password: "dmFsdWUtMg0KDQo="
```

1

The allowable format for the keys in the **data** field must meet the guidelines in the **DNS_SUBDOMAIN** value in [the Kubernetes identifiers glossary](#).

17.2. PROPERTIES OF SECRETS

Key properties include:

- ✧ Secret data can be referenced independently from its definition.
- ✧ Secret data never comes to rest on the node. Volumes are backed by temporary file-storage facilities (tmpfs).
- ✧ Secret data can be shared within a namespace.

17.2.1. Secrets and the Pod Lifecycle

A secret must be created before the pods that depend on it.

Containers read the secret from the files. If a secret is expected to be stored in an environment variable, then you must modify the image to populate the environment variable from the file before running the main program.

Once a pod is created, its secret volumes do not change, even if the secret resource is modified. To change the secret used, the original pod must be deleted, and a new pod (perhaps with an identical PodSpec) must be created. An exception to this is when a node is rebooted and the secret data must

be re-read from the API server. Updating a secret follows the same workflow as deploying a new container image. The `kubectl rollingupdate` command can be used.

The **resourceVersion** value in a secret is not specified when it is referenced. Therefore, if a secret is updated at the same time as pods are starting, then the version of the secret will be used for the pod will not be defined.



Note

Currently, it is not possible to check the resource version of a secret object that was used when a pod was created. It is planned that pods will report this information, so that a controller could restart ones using a old **resourceVersion**. In the interim, do not update the data of existing secrets, but create new ones with distinct names.

17.3. CREATING AND USING SECRETS

When creating secrets:

- ✎ Create a secret object with secret data
- ✎ Create a pod with a volume of type **secret** and a container to mount the volume
- ✎ Update the pod's service account to allow the reference to the secret.

17.3.1. Creating Secrets

To create a secret object, use the following command, where the JSON file is a predefined secret:

```
$ oc create -f secret.json
```

17.3.2. Secrets in Volumes and Environment Variables

See [examples](#) of YAML files with secret data.

After you [create a secret](#), you can:

1. Create the pod to reference your secret:

```
$ oc create -f <your_yaml_file>.yaml
```

2. Get the logs:

```
$ oc logs secret-example-pod
```

3. Delete the pod:

```
$ oc delete pod secret-example-pod
```

17.3.3. Image Pull Secrets

See [Using Image Pull Secrets](#) for more information.

17.3.4. Source Clone Secrets

See [Using Private Repositories for Builds](#) for more information.

17.4. RESTRICTIONS

To use a secret, a pod needs to reference the secret. A secret can be used with a pod in two ways: either as files in a volume mounted on one or more of its containers, or used by kubelet when pulling images for the pod.

Volume type secrets write data into the container as a file using the volume mechanism.

imagePullSecrets use service accounts for the automatic injection of the secret into all pods in a namespace.

When a template contains a secret definition, the only way for the template to use the provided secret is to ensure that the secret volume sources are validated and that the specified object reference actually points to an object of type **Secret**. Therefore, a secret needs to be created before any pods that depend on it. The most effective way to ensure this is to have it get injected automatically through the use of a service account.

Secret API objects reside in a namespace. They can only be referenced by pods in that same namespace.

Individual secrets are limited to 1MB in size. This is to discourage the creation of large secrets that would exhaust apiserver and kubelet memory. However, creation of a number of smaller secrets could also exhaust memory.

17.4.1. Secret Data Keys

Secret keys must be in a DNS subdomain.

17.5. EXAMPLES

Example 17.2. YAML Secret That Will Create Four Files

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: dmFsdWUtMQ0K 1
  password: dmFsdWUtMQ0KDQo= 2
stringData:
  hostname: myapp.mydomain.com 3
secret.properties: |- 4
  property1=valueA
  property2=valueB
```

File contains decoded values.

2

File contains decoded values.

3

File contains the provided string.

4

File contains the provided data.

Example 17.3. YAML of a Pod Populating Files in a Volume with Secret Data

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "cat /etc/secret-volume/*" ]
      volumeMounts:
        # name must match the volume name below
        - name: secret-volume
          mountPath: /etc/secret-volume
          readOnly: true
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret
      restartPolicy: Never
```

Example 17.4. YAML of a Pod Populating Environment Variables with Secret Data

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "export" ]
```

```
env:
  - name: TEST_SECRET_USERNAME_ENV_VAR
    valueFrom:
      secretKeyRef:
        name: test-secret
        key: username
restartPolicy: Never
```

17.6. TROUBLESHOOTING

Table 17.1. Troubleshooting Guidance for Secrets

Issue	Resolution
<p>A service certificate generation fails with (service's service.alpha.openshift.io/serving-cert-generation-error annotation contains):</p> <pre>secret/ssl-key reference s serviceUID D 62ad25ca- d703- 11e6- 9d6f- 0e9c0057b 608, which does not match 77b6dd80- d716- 11e6- 9d6f- 0e9c0057b 60</pre>	<p>The service that generated the certificate no longer exists (has different serviceUID). You must force certificates regeneration by removing the old secret, and clearing following annotations on the service service.alpha.openshift.io/serving-cert-generation-error, service.alpha.openshift.io/serving-cert-generation-error-num:</p> <pre>\$ oc delete secret <secret_name> \$ oc annotate service <service_name> service.alpha.openshift.io/serving-cert-generation- error- \$ oc annotate service <service_name> service.alpha.openshift.io/serving-cert-generation- error-num-</pre> <div>Note<p>The command removing annotation has a - after the annotation name to be removed.</p></div>

CHAPTER 18. CONFIGMAPS

18.1. OVERVIEW

Many applications require configuration using some combination of configuration files, command line arguments, and environment variables. These configuration artifacts should be decoupled from image content in order to keep containerized applications portable.

The **ConfigMap** object provides mechanisms to inject containers with configuration data while keeping containers agnostic of OpenShift Enterprise. A **ConfigMap** can be used to store fine-grained information like individual properties or coarse-grained information like entire configuration files or JSON blobs.

The **ConfigMap** API object holds key-value pairs of configuration data that can be consumed in pods or used to store configuration data for system components such as controllers. **ConfigMap** is similar to [secrets](#), but designed to more conveniently support working with strings that do not contain sensitive information.

For example:

Example 18.1. ConfigMap Object Definition

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data: 1
  example.property.1: hello
  example.property.2: world
  example.property.file: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
```

1

Contains the configuration data.

Configuration data can be consumed in pods in a variety of ways. A **ConfigMap** can be used to:

1. Populate the value of environment variables.
2. Set command-line arguments in a container.
3. Populate configuration files in a volume.

Both users and system components may store configuration data in a **ConfigMap**.

18.2. CREATING CONFIGMAPS

You can use the following command to create a **ConfigMap** easily from directories, specific files, or literal values:

```
$ oc create configmap <configmap_name> [options]
```

The following sections cover the different ways you can create a **ConfigMap**.

18.2.1. Creating from Directories

Consider a directory with some files that already contain the data with which you want to populate a **ConfigMap**:

```
$ ls example-files
game.properties
ui.properties

$ cat example-files/game.properties
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30

$ cat example-files/ui.properties
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

You can use the following command to create a **ConfigMap** holding the content of each file in this directory:

```
$ oc create configmap game-config \
  --from-file=example-files/
```

When the **--from-file** option points to a directory, each file directly in that directory is used to populate a key in the **ConfigMap**, where the name of the key is the file name, and the value of the key is the content of the file.

For example, the above command creates the following **ConfigMap**:

```
$ oc describe configmaps game-config
Name:          game-config
Namespace:     default
Labels:        <none>
Annotations:   <none>

Data
```

```
game.properties:      121 bytes
ui.properties:        83 bytes
```

You can see the two keys in the map are created from the file names in the directory specified in the command. Because the content of those keys may be large, the output of **oc describe** only shows the names of the keys and their sizes.

If you want to see the values of the keys, you can **oc get** the object with the **-o** option:

```
$ oc get configmaps game-config -o yaml

apiVersion: v1
data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:34:05Z
  name: game-config
  namespace: default
  resourceVersion: "407"-
  selflink: /api/v1/namespaces/default/configmaps/game-config
  uid: 30944725-d66e-11e5-8cd0-68f728db1985
```

18.2.2. Creating from Files

You can also pass the **--from-file** option with a specific file, and pass it multiple times to the CLI. The following yields equivalent results to the [Creating from Directories](#) example:

1. Create the **ConfigMap** specifying a specific file:

```
$ oc create configmap game-config-2 \
  --from-file=example-files/game.properties \
  --from-file=example-files/ui.properties
```

2. Verify the results:

```
$ oc get configmaps game-config-2 -o yaml

apiVersion: v1
```



```

data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z
  name: game-config-2
  namespace: default
  resourceVersion: "516"
  selflink: /api/v1/namespaces/default/configmaps/game-config-2
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985

```

You can also set the key to use for an individual file with the **--from-file** option by passing an expression of **key=value**. For example:

1. Create the **ConfigMap** specifying a key-value pair:

```

$ oc create configmap game-config-3 \
  --from-file=game-special-key=example-files/game.properties

```

2. Verify the results:

```

$ oc get configmaps game-config-3 -o yaml

apiVersion: v1
data:
  game-special-key: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  selflink: /api/v1/namespaces/default/configmaps/game-config-3
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985

```

18.2.3. Creating from Literal Values

You can also supply literal values for a **ConfigMap**. The **--from-literal** option takes a **key=value** syntax that allows literal values to be supplied directly on the command line:

1. Create the **ConfigMap** specifying a literal value:

```
$ oc create configmap special-config \
  --from-literal=special.how=very \
  --from-literal=special.type=charm
```

2. Verify the results:

```
$ oc get configmaps special-config -o yaml

apiVersion: v1
data:
  special.how: very
  special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  selflink: /api/v1/namespaces/default/configmaps/special-config
  uid: dadce046-d673-11e5-8cd0-68f728db1985
```

18.3. USE CASES: CONSUMING CONFIGMAPS IN PODS

The following sections describe some uses cases when consuming **ConfigMap** objects in pods.

18.3.1. Consuming in Environment Variables

A **ConfigMap** can be used to populate the value of command line arguments. For example, consider the following **ConfigMap**:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

You can consume the keys of this **ConfigMap** in a pod using **configMapKeyRef** sections:

```
apiVersion: v1
```

```

kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
  - name: test-container
    image: gcr.io/google_containers/busybox
    command: [ "/bin/sh", "-c", "env" ]
    env:
      - name: SPECIAL_LEVEL_KEY
        valueFrom:
          configMapKeyRef:
            name: special-config
            key: special.how
      - name: SPECIAL_TYPE_KEY
        valueFrom:
          configMapKeyRef:
            name: special-config
            key: special.type
    restartPolicy: Never

```

When this pod is run, its output will include the following lines:

```

SPECIAL_LEVEL_KEY=very
SPECIAL_TYPE_KEY=charm

```

18.3.2. Setting Command-line Arguments

A **ConfigMap** can also be used to set the value of the command or arguments in a container. This is accomplished using the Kubernetes substitution syntax **\$(VAR_NAME)**. Consider the following **ConfigMap**:

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm

```

To inject values into the command line, you must consume the keys you want to use as environment variables, as in the [Consuming in Environment Variables](#) use case. Then you can refer to them in a container's command using the **\$(VAR_NAME)** syntax.

```

apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
  - name: test-container

```

```

    image: gcr.io/google_containers/busybox
    command: [ "/bin/sh", "-c", "echo ${SPECIAL_LEVEL_KEY}
${SPECIAL_TYPE_KEY}" ]
    env:
      - name: SPECIAL_LEVEL_KEY
        valueFrom:
          configMapKeyRef:
            name: special-config
            key: special.how
      - name: SPECIAL_TYPE_KEY
        valueFrom:
          configMapKeyRef:
            name: special-config
            key: special.type
    restartPolicy: Never

```

When this pod is run, the output from the **test-container** container will be:

```
very charm
```

18.3.3. Consuming in Volumes

A **ConfigMap** can also be consumed in volumes. Returning again to the following example **ConfigMap**:

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm

```

You have a couple different options for consuming this **ConfigMap** in a volume. The most basic way is to populate the volume with files where the key is the file name and the content of the file is the value of the key:

```

apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "cat", "/etc/config/special.how" ]
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:

```

```
- name: config-volume
  configMap:
    name: special-config
  restartPolicy: Never
```

When this pod is run, the output will be:

```
very
```

You can also control the paths within the volume where **ConfigMap** keys are projected:

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "cat", "/etc/config/path/to/special-key" ]
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
          - key: special.how
            path: path/to/special-key
      restartPolicy: Never
```

When this pod is run, the output will be:

```
very
```

18.4. EXAMPLE: CONFIGURING REDIS

For a real-world example, you can configure Redis using a **ConfigMap**. To inject Redis with the recommended configuration for using Redis as a cache, the Redis configuration file should contain the following:

```
maxmemory 2mb
maxmemory-policy allkeys-lru
```

If your configuration file is located at **example-files/redis/redis-config**, create a **ConfigMap** with it:

1. Create the **ConfigMap** specifying the configuration file:

```
$ oc create configmap example-redis-config \
  --from-file=example-files/redis/redis-config
```

2. Verify the results:

```
$ oc get configmap example-redis-config -o yaml

apiVersion: v1
data:
  redis-config: |
    maxmemory 2mb
    maxmemory-policy allkeys-lru
kind: ConfigMap
metadata:
  creationTimestamp: 2016-04-06T05:53:07Z
  name: example-redis-config
  namespace: default
  resourceVersion: "2985"
  selflink: /api/v1/namespaces/default/configmaps/example-redis-
config
  uid: d65739c1-fbbb-11e5-8a72-68f728db1985
```

Now, create a pod that uses this **ConfigMap**:

1. Create a pod definition like the following and save it to a file, for example *redis-pod.yaml*:

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: kubernetes/redis:v1
    env:
    - name: MASTER
      value: "true"
    ports:
    - containerPort: 6379
    resources:
      limits:
        cpu: "0.1"
    volumeMounts:
    - mountPath: /redis-master-data
      name: data
    - mountPath: /redis-master
      name: config
  volumes:
  - name: data
    emptyDir: {}
  - name: config
    configMap:
```

```
name: example-redis-config
items:
- key: redis-config
  path: redis.conf
```

2. Create the pod:

```
$ oc create -f redis-pod.yaml
```

The newly-created pod has a **ConfigMap** volume that places the **redis-config** key of the **example-redis-config ConfigMap** into a file called **redis.conf**. This volume is mounted into the **/redis-master** directory in the Redis container, placing our configuration file at **/redis-master/redis.conf**, which is where the image looks for the Redis configuration file for the master.

If you **oc exec** into this pod and run the **redis-cli** tool, you can check that the configuration was applied correctly:

```
$ oc exec -it redis redis-cli
127.0.0.1:6379> CONFIG GET maxmemory
1) "maxmemory"
2) "2097152"
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "allkeys-lru"
```

18.5. RESTRICTIONS

A **ConfigMap** must be created before they are consumed in pods. Controllers can be written to tolerate missing configuration data; consult individual components configured via **ConfigMap** on a case-by-case basis.

ConfigMap objects reside in a project. They can only be referenced by pods in the same project.

The Kubelet only supports use of a **ConfigMap** for pods it gets from the API server. This includes any pods created using the CLI, or indirectly from a replication controller. It does not include pods created using the OpenShift Enterprise node's **--manifest-url** flag, its **--config** flag, or its REST API (these are not common ways to create pods).

CHAPTER 19. USING DAEMONSETS

19.1. OVERVIEW

A daemonset can be used to run replicas of a pod on specific or all nodes in an OpenShift Enterprise cluster.

Use daemonsets to create shared storage, run a logging pod on every node in your cluster, or deploy a monitoring agent on every node.

For more information on daemonsets, see the [Kubernetes documentation](#).

19.2. CREATING DAEMONSETS



Important

Before creating daemonsets, ensure you have been [given the required role by your OpenShift Enterprise administrator](#).

When creating daemonsets, the **nodeSelector** field is used to indicate the nodes on which the daemonset should deploy replicas.

1. Define the daemonset yaml file:

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: hello-daemonset
spec:
  selector:
    matchLabels:
      name: hello-daemonset ①
  template:
    metadata:
      labels:
        name: hello-daemonset ②
    spec:
      nodeSelector: ③
      type: infra
      containers:
      - image: openshift/hello-openshift
        imagePullPolicy: Always
        name: registry
        ports:
        - containerPort: 80
          protocol: TCP
        resources: {}
        terminationMessagePath: /dev/termination-log
      serviceAccount: default
      terminationGracePeriodSeconds: 10
```


1

The label selector that determines which pods belong to the daemonset.

2

The pod template's label selector. Must match the label selector above.

3

The node selector that determines on which nodes pod replicas should be deployed.

2. Create the daemonset object:

```
oc create -f daemonset.yaml
```

3. To verify that the pods were created, and that each node has a pod replica:

- a. Find the daemonset pods:

```
$ oc get pods
hello-daemonset-cx6md    1/1      Running    0          2m
hello-daemonset-e3md9    1/1      Running    0          2m
```

- b. View the pods to verify the pod has been placed onto the node:

```
$ oc describe pod/hello-daemonset-cx6md | grep Node
Node:          openshift-node01.hostname.com/10.14.20.134
$ oc describe pod/hello-daemonset-e3md9 | grep Node
Node:          openshift-node02.hostname.com/10.14.20.137
```

Important

Currently, updating a daemonset's pod template does not affect existing pod replicas. Moreover, if you delete a daemonset and then create a new daemonset with a different template but the same label selector, it will recognize any existing pod replicas as having matching labels and thus will not update them or create new replicas despite a mismatch in the pod template.

To update a daemonset, force new pod replicas to be created by deleting the old replicas or nodes.

CHAPTER 20. POD AUTOSCALING

20.1. OVERVIEW

A horizontal pod autoscaler, defined by a **HorizontalPodAutoscaler** object, specifies how the system should automatically increase or decrease the scale of a replication controller or deployment configuration, based on metrics collected from the pods that belong to that replication controller or deployment configuration.

**Note**

Horizontal pod autoscaling is supported starting in OpenShift Enterprise 3.1.1.

20.2. REQUIREMENTS FOR USING HORIZONTAL POD AUTOSCALERS

In order to use horizontal pod autoscalers, your cluster administrator must have [properly configured cluster metrics](#).

20.3. SUPPORTED METRICS

The following metrics are supported by horizontal pod autoscalers:

Table 20.1. Metrics

Metric	Description
CPU Utilization	Percentage of the requested CPU

20.4. AUTOSCALING

You can create a horizontal pod autoscaler with the **oc autoscale** command and specify the minimum and maximum number of pods you want to run, as well as the CPU utilization your pods should target.

After a horizontal pod autoscaler is created, it begins attempting to query Heapster for metrics on the pods. It may take one to two minutes before Heapster obtains the initial metrics.

After metrics are available in Heapster, the horizontal pod autoscaler computes the ratio of the current metric utilization with the desired metric utilization, and scales up or down accordingly. The scaling will occur at a regular interval, but it may take one to two minutes before metrics make their way into Heapster.

For replication controllers, this scaling corresponds directly to the replicas of the replication controller. For deployment configurations, scaling corresponds directly to the replica count of the deployment configuration. Note that autoscaling applies only to the latest deployment in the **Complete** phase.

20.5. CREATING A HORIZONTAL POD AUTOSCALER

Use the **oc autoscale** command and specify at least the maximum number of pods you want to run at any given time. You can optionally specify the minimum number of pods and the average CPU utilization your pods should target, otherwise those are given default values from the OpenShift Enterprise server.

For example:

```
$ oc autoscale dc/frontend --min 1 --max 10 --cpu-percent=80
deploymentconfig "frontend" autoscaled
```

The above example creates a horizontal pod autoscaler with the following definition:

Example 20.1. Horizontal Pod Autoscaler Object Definition

```
apiVersion: extensions/v1beta1
kind: HorizontalPodAutoscaler
metadata:
  name: frontend 1
spec:
  scaleRef:
    kind: DeploymentConfig 2
    name: frontend 3
    apiVersion: v1 4
    subresource: scale
  minReplicas: 1 5
  maxReplicas: 10 6
  cpuUtilization:
    targetPercentage: 80 7
```

1

The name of this horizontal pod autoscaler object

2

The kind of object to scale

3

The name of the object to scale

4

The API version of the object to scale

5

The minimum number of replicas to which to scale down

6

The maximum number of replicas to which to scale up

7

The percentage of the requested CPU that each pod should ideally be using

20.6. VIEWING A HORIZONTAL POD AUTOSCALER

To view the status of a horizontal pod autoscaler:

```
$ oc get hpa/frontend
NAME          REFERENCE                                     TARGET
CURRENT      MINPODS      MAXPODS      AGE
frontend      DeploymentConfig/default/frontend/scale      80%
79%           1            10           8d

$ oc describe hpa/frontend
Name:          frontend
Namespace:     default
Labels:        <none>
CreationTimestamp: Mon, 26 Oct 2015 21:13:47 -0400
Reference:
DeploymentConfig/default/frontend/scale
Target CPU utilization:      80%
Current CPU utilization:     79%
Min pods:                    1
Max pods:                     10
```

CHAPTER 21. MANAGING VOLUMES

21.1. OVERVIEW

Containers are not persistent by default; on restart, their contents are cleared. Volumes are mounted file systems available to [pods](#) and their containers which may be backed by a number of host-local or network attached storage endpoints.

To ensure that the file system on the volume contains no errors and, if errors are present, to repair them when possible, OpenShift Enterprise invokes the **fsck** utility prior to the **mount** utility. This occurs when either adding a volume or updating an existing volume.

The simplest volume type is **EmptyDir**, which is a temporary directory on a single machine. Administrators may also allow you to request a [persistent volume](#) that is automatically attached to your pods.



Note

EmptyDir volume storage may be restricted by a quota based on the pod's FSGroup, if enabled by your cluster administrator.

You can use the CLI command **oc volume** to [add](#), [update](#), or [remove](#) volumes and volume mounts for any object that has a pod template like [replication controllers](#) or [deployment configurations](#). You can also [list](#) volumes in pods or any object that has a pod template.

21.2. GENERAL CLI USAGE

The **oc volume** command uses the following general syntax:

```
$ oc volume <object_selection> <operation> <mandatory_parameters>
<optional_parameters>
```

This topic uses the form **<object_type>/<name>** for **<object_selection>** in later examples. However, you can choose one of the following options:

Table 21.1. Object Selection

Syntax	Description	Example
<object_type> <name>	Selects <name> of type <object_type> .	deploymentConfig registry
<object_type>/<name>	Selects <name> of type <object_type> .	deploymentConfig/registry

Syntax	Description	Example
<object_type>--selector=<object_label_selector>	Selects resources of type <object_type> that matched the given label selector.	deploymentConfig--selector="name=registry"
<object_type> --all	Selects all resources of type <object_type> .	deploymentConfig --all
-f or --filename=<file_name>	File name, directory, or URL to file to use to edit the resource.	-f registry-deployment-config.json

The **<operation>** can be one of **--add**, **--remove**, or **--list**.

Any **<mandatory_parameters>** or **<optional_parameters>** are specific to the selected operation and are discussed in later sections.

21.3. ADDING VOLUMES

To add a volume, a volume mount, or both to pod templates:

```
$ oc volume <object_type>/<name> --add [options]
```

Table 21.2. Supported Options for Adding Volumes

Option	Description	Default
--name	Name of the volume.	Automatically generated, if not specified.
-t, --type	Name of the volume source. Supported values: emptyDir , hostPath , secret , configmap , or persistentVolumeClaim .	emptyDir
-c, --containers	Select containers by name. It can also take wildcard '*' that matches any character.	'*'

Option	Description	Default
-m, --mount-path	Mount path inside the selected containers.	
--path	Host path. Mandatory parameter for --type=hostPath .	
--secret-name	Name of the secret. Mandatory parameter for --type=secret .	
--claim-name	Name of the persistent volume claim. Mandatory parameter for --type=persistentVolumeClaim .	
--source	Details of volume source as a JSON string. Recommended if the desired volume source is not supported by --type . See available volume sources	
-o, --output	Display the modified objects instead of updating them on the server. Supported values: json , yaml .	
--output-version	Output the modified objects with the given version.	api-version

Examples

Add a new volume source **emptyDir** to deployment configuration **registry**:

```
$ oc volume dc/registry --add
```

Add volume **v1** with secret **\$secret** for replication controller **r1** and mount inside the containers at **/data**:

```
$ oc volume rc/r1 --add --name=v1 --type=secret --secret-name='$secret'
--mount-path=/data
```

Add existing persistent volume **v1** with claim name **pvc1** to deployment configuration **dc.json** on disk, mount the volume on container **c1** at **/data**, and update the deployment configuration on the server:

```
$ oc volume -f dc.json --add --name=v1 --type=persistentVolumeClaim \
  --claim-name=pvc1 --mount-path=/data --containers=c1
```

Add volume **v1** based on Git repository <https://github.com/namespace1/project1> with revision **5125c45f9f563** for all replication controllers:

```
$ oc volume rc --all --add --name=v1 \
  --source='{ "gitRepo": {
    "repository":
      "https://github.com/namespace1/project1",
    "revision": "5125c45f9f563"
  } }'
```

21.4. UPDATING VOLUMES

Updating existing volumes or volume mounts is the same as [adding volumes](#), but with the **--overwrite** option:

```
$ oc volume <object_type>/<name> --add --overwrite [options]
```

Examples

Replace existing volume **v1** for replication controller **r1** with existing persistent volume claim **pvc1**:

```
$ oc volume rc/r1 --add --overwrite --name=v1 --
  type=persistentVolumeClaim --claim-name=pvc1
```

Change deployment configuration **d1** mount point to **/opt** for volume **v1**:

```
$ oc volume dc/d1 --add --overwrite --name=v1 --mount-path=/opt
```

21.5. REMOVING VOLUMES

To remove a volume or volume mount from pod templates:

```
$ oc volume <object_type>/<name> --remove [options]
```

Table 21.3. Supported Options for Removing Volumes

Option	Description	Default
--name	Name of the volume.	

Option	Description	Default
-c, --containers	Select containers by name. It can also take wildcard '*' that matches any character.	'*'
--confirm	Indicate that you want to remove multiple volumes at once.	
-o, --output	Display the modified objects instead of updating them on the server. Supported values: json , yaml .	
--output-version	Output the modified objects with the given version.	api-version

Some examples:

Remove a volume **v1** from deployment config **d1**:

```
$ oc volume dc/d1 --remove --name=v1
```

Unmount volume **v1** from container **c1** for deployment configuration **d1** and remove the volume **v1** if it is not referenced by any containers on **d1**:

```
$ oc volume dc/d1 --remove --name=v1 --containers=c1
```

Remove all volumes for replication controller **r1**:

```
$ oc volume rc/r1 --remove --confirm
```

21.6. LISTING VOLUMES

To list volumes or volume mounts for pods or pod templates:

```
$ oc volume <object_type>/<name> --list [options]
```

List volume supported options:

Option	Description	Default
--name	Name of the volume.	

Option	Description	Default
-c, --containers	Select containers by name. It can also take wildcard '*' that matches any character.	'*'

Examples

List all volumes for pod **p1**:

```
$ oc volume pod/p1 --list
```

List volume **v1** defined on all deployment configurations:

```
$ oc volume dc --all --name=v1
```

CHAPTER 22. USING PERSISTENT VOLUMES

22.1. OVERVIEW

A **PersistentVolume** object is a storage resource in an OpenShift Enterprise cluster. Storage is provisioned by your cluster administrator by creating **PersistentVolume** objects from sources such as GCE Persistent Disk, AWS Elastic Block Store (EBS), and NFS mounts.



Note

The [Installation and Configuration Guide](#) provides instructions for cluster administrators on provisioning an OpenShift Enterprise cluster with persistent storage using [NFS](#), [GlusterFS](#), [Ceph RBD](#), [OpenStack Cinder](#), [AWS EBS](#), [GCE Persistent Disk](#), [iSCSI](#), and [Fibre Channel](#).

Storage can be made available to you by laying claims to the resource. You can make a request for storage resources using a **PersistentVolumeClaim** object; the claim is paired with a volume that generally matches your request.

22.2. REQUESTING STORAGE

You can request storage by creating **PersistentVolumeClaim** objects in your projects:

Persistent Volume Claim Object Definition

```
apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
  name: "claim1"
spec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: "1Gi"
  volumeName: "pv0001"
```

22.3. VOLUME AND CLAIM BINDING

A **PersistentVolume** is a specific resource. A **PersistentVolumeClaim** is a request for a resource with specific attributes, such as storage size. In between the two is a process that matches a claim to an available volume and binds them together. This allows the claim to be used as a volume in a pod. OpenShift Enterprise finds the volume backing the claim and mounts it into the pod.

You can tell whether a claim or volume is bound by querying using the CLI:

```
$ oc get pvc
NAME          LABELS          STATUS          VOLUME
claim1        map[]           Bound           pv0001
```

```
$ oc get pv
NAME                                LABELS                                CAPACITY
ACCESSMODES                        STATUS    CLAIM
pv0001                             map[]    5368709120    RW0
Bound    yournamespace / claim1
```

22.4. CLAIMS AS VOLUMES IN PODS

A **PersistentVolumeClaim** is used by a pod as a volume. OpenShift Enterprise finds the claim with the given name in the same namespace as the pod, then uses the claim to find the corresponding volume to mount.

Pod Definition with a Claim

```
apiVersion: "v1"
kind: "Pod"
metadata:
  name: "mypod"
  labels:
    name: "frontendhttp"
spec:
  containers:
    -
      name: "myfrontend"
      image: "nginx"
      ports:
        -
          containerPort: 80
          name: "http-server"
      volumeMounts:
        -
          mountPath: "/var/www/html"
          name: "pvol"
  volumes:
    -
      name: "pvol"
      persistentVolumeClaim:
        claimName: "claim1"
```

22.5. VOLUME AND CLAIM PRE-BINDING

If you know exactly what **PersistentVolume** you want your **PersistentVolumeClaim** to bind to, you can specify the PV in your PVC using the **volumeName** field. This method skips the normal matching and binding process. The PVC will only be able to bind to a PV that has the same name specified in **volumeName**. If such a PV with that name exists and is **Available**, the PV and PVC will be bound regardless of whether the PV satisfies the PVC's label selector, access modes, and resource requests.

Example 22.1. Persistent Volume Claim Object Definition with volumeName

■

```

apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
  name: "claim1"
spec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: "1Gi"
  volumeName: "pv0001"

```

Important

The ability to set **claimRefs** is a temporary workaround for the described use cases. A long-term solution for limiting who can claim a volume is in development.

Note

The cluster administrator should first consider configuring selector-label volume binding before resorting to setting **claimRefs** on behalf of users.

You may also want your cluster administrator to "reserve" the volume for only your claim so that nobody else's claim can bind to it before yours does. In this case, the administrator can specify the PVC in the PV using the **claimRef** field. The PV will only be able to bind to a PVC that has the same name and namespace specified in **claimRef**. The PVC's access modes and resource requests must still be satisfied in order for the PV and PVC to be bound, though the label selector is ignored.

Persistent Volume Object Definition with claimRef

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  nfs:
    path: /tmp
    server: 172.17.0.2
  persistentVolumeReclaimPolicy: Recycle
  claimRef:
    name: claim1
    namespace: default

```

Specifying a **volumeName** in your PVC does not prevent a different PVC from binding to the specified PV before yours does. Your claim will remain **Pending** until the PV is **Available**.

Specifying a **claimRef** in a PV does not prevent the specified PVC from being bound to a different PV. The PVC is free to choose another PV to bind to according to the normal binding process. Therefore, to avoid these scenarios and ensure your claim gets bound to the volume you want, you must ensure that both **volumeName** and **claimRef** are specified.

You can tell that your setting of **volumeName** and/or **claimRef** influenced the matching and binding process by inspecting a **Bound** PV and PVC pair for the **pv.kubernetes.io/bound-by-controller** annotation. The PVs and PVCs where you set the **volumeName** and/or **claimRef** yourself will have no such annotation, but ordinary PVs and PVCs will have it set to **"yes"**.

When a PV has its **claimRef** set to some PVC name and namespace, and is reclaimed according to a **Retain** or **Recycle** recycling policy, its **claimRef** will remain set to the same PVC name and namespace even if the PVC or the whole namespace no longer exists.

CHAPTER 23. EXECUTING REMOTE COMMANDS

23.1. OVERVIEW

You can use the CLI to execute remote commands in a container. This allows you to run general Linux commands for routine operations in the container.



Important

For security purposes, the **oc exec** command does not work when accessing privileged containers. See the [CLI operations topic](#) for more information.

23.2. BASIC USAGE

Support for remote container command execution is built into [the CLI](#):

```
$ oc exec <pod> [-c <container>] <command> [<arg_1> ... <arg_n>]
```

For example:

```
$ oc exec mypod date
Thu Apr 9 02:21:53 UTC 2015
```

23.3. PROTOCOL

Clients initiate the execution of a remote command in a container by issuing a request to the Kubernetes API server:

```
/proxy/minions/<node_name>/exec/<namespace>/<pod>/<container>?command=
<command>
```

In the above URL:

- ✧ **<node_name>** is the FQDN of the node.
- ✧ **<namespace>** is the namespace of the target pod.
- ✧ **<pod>** is the name of the target pod.
- ✧ **<container>** is the name of the target container.
- ✧ **<command>** is the desired command to be executed.

For example:

```
/proxy/minions/node123.openshift.com/exec/myns/mypod/mycontainer?
command=date
```

Additionally, the client can add parameters to the request to indicate if:

- ✧ the client should send input to the remote container's command (stdin).
- ✧ the client's terminal is a TTY.
- ✧ the remote container's command should send output from stdout to the client.
- ✧ the remote container's command should send output from stderr to the client.

After sending an **exec** request to the API server, the client upgrades the connection to one that supports multiplexed streams; the current implementation uses **SPDY**.

The client creates one stream each for stdin, stdout, and stderr. To distinguish among the streams, the client sets the **streamType** header on the stream to one of **stdin**, **stdout**, or **stderr**.

The client closes all streams, the upgraded connection, and the underlying connection when it is finished with the remote command execution request.



Note

Administrators can see the [Architecture](#) guide for more information.

CHAPTER 24. COPYING FILES TO OR FROM A CONTAINER

24.1. OVERVIEW

You can use the CLI to copy local files to or from a remote directory in a container. This is a useful tool for copying database archives to and from your pods for backup and restore purposes. It can also be used to copy source code changes into a running pod for development debugging, when the running pod supports hot reload of source files.

24.2. BASIC USAGE

Support for copying local files to or from a container is built into [the CLI](#):

```
$ oc rsync <source> <destination> [-c <container>]
```

For example, to copy a local directory to a pod directory:

```
$ oc rsync /home/user/source devpod1234:/src
```

Or to copy a pod directory to a local directory:

```
$ oc rsync devpod1234:/src /home/user/source
```

24.3. BACKING UP AND RESTORING DATABASES

Use **oc rsync** to copy database archives from an existing database container to a new database container's persistent volume directory.



Note

MySQL is used in the example below. Replace **mysql** | **MYSQL** with **pgsql** | **PGSQL** or **mongodb** | **MONGODB** and refer to [the migration guide](#) to find the exact commands for each of our supported database images. The example assumes an existing database container.

1. Back up the existing database from a running database pod:

```
$ oc rsh <existing db container>
# mkdir /var/lib/mysql/data/db_archive_dir
# mysqldump --skip-lock-tables -h ${MYSQL_SERVICE_HOST} -P
${MYSQL_SERVICE_PORT} \
-u ${MYSQL_USER} --password="${MYSQL_PASSWORD}" --all-databases >
/var/lib/mysql/data/db_archive_dir/all.sql
# exit
```

2. Remote sync the archive file to your local machine:

```
$ oc rsync <existing db container with db
archive>:/var/lib/mysql/data/db_archive_dir /tmp/.
```

3. Start a second MySQL pod into which to load the database archive file created above. The MySQL pod must have a unique **DATABASE_SERVICE_NAME**.

```
$ oc new-app mysql-persistent \
  -p MYSQL_USER=<archived mysql username> \
  -p MYSQL_PASSWORD=<archived mysql password> \
  -p MYSQL_DATABASE=<archived database name> \
  -p DATABASE_SERVICE_NAME='mysql2' 1
$ oc rsync /tmp/db_archive_dir new_dbpod1234:/var/lib/mysql/data
$ oc rsh new_dbpod1234
```

1

mysql is the default. In this example, **mysql2** is created.

4. Use the appropriate commands to restore the database in the new database container from the copied database archive directory:

MySQL

```
$ cd /var/lib/mysql/data/db_archive_dir
$ mysql -u root
$ source all.sql
$ GRANT ALL PRIVILEGES ON <dbname>.* TO '<your
username>'@'localhost'; FLUSH PRIVILEGES;
$ cd ../; rm -rf /var/lib/mysql/data/db_backup_dir
```

You now have two MySQL database pods running in your project with the archived database.

24.4. REQUIREMENTS

The **oc rsync** command uses the local **rsync** command if present on the client's machine. This requires that the remote container also have the **rsync** command.

If **rsync** is not found locally or in the remote container, then a tar archive will be created locally and sent to the container where **tar** will be used to extract the files. If **tar** is not available in the remote container, then the copy will fail.

The **tar** copy method does not provide the same functionality as **rsync**. For example, **rsync** creates the destination directory if it does not exist and will only send files that are different between the source and the destination.

**Note**

In Windows, the **cwRsync** client should be installed and added to the PATH for use with the **oc rsync** command.

24.5. SPECIFYING THE COPY SOURCE

The source argument of the **oc rsync** command must point to either a local directory or a pod directory. Individual files are not currently supported.

When specifying a pod directory the directory name must be prefixed with the pod name:

```
<pod name>:<dir>
```

Just as with UNIX **rsync**, if the directory name ends in a path separator (/), only the contents of the directory are copied to the destination. Otherwise, the directory itself is copied to the destination with all its contents.

24.6. SPECIFYING THE COPY DESTINATION

The destination argument of the **oc rsync** command must point to a directory. If the directory does not exist, but **rsync** is used for copy, the directory is created for you.

24.7. DELETING FILES AT THE DESTINATION

The **--delete** flag may be used to delete any files in the remote directory that are not in the local directory.

CHAPTER 25. PORT FORWARDING

25.1. OVERVIEW

You can use the CLI to forward one or more local ports to a pod. This allows you to listen on a given or random port locally, and have data forwarded to and from given ports in the pod.

25.2. BASIC USAGE

Support for port forwarding is built into [the CLI](#):

```
$ oc port-forward <pod> [<local_port>:]<remote_port> [...  
[<local_port_n>:]<remote_port_n>]
```

The CLI listens on each local port specified by the user, forwarding via the [protocol](#) described below.

Ports may be specified using the following formats:

5000	The client listens on port 5000 locally and forwards to 5000 in the pod.
6000:5000 0	The client listens on port 6000 locally and forwards to 5000 in the pod.
:5000 or 0:5000	The client selects a free local port and forwards to 5000 in the pod.

For example, to listen on ports **5000** and **6000** locally and forward data to and from ports **5000** and **6000** in the pod, run:

```
$ oc port-forward <pod> 5000 6000
```

To listen on port **8888** locally and forward to **5000** in the pod, run:

```
$ oc port-forward <pod> 8888:5000
```

To listen on a free port locally and forward to **5000** in the pod, run:

```
$ oc port-forward <pod> :5000
```

Or, alternatively:

```
$ oc port-forward <pod> 0:5000
```

25.3. PROTOCOL

Clients initiate port forwarding to a pod by issuing a request to the Kubernetes API server:

```
/proxy/minions/<node_name>/portForward/<namespace>/<pod>
```

In the above URL:

- ✱ **<node_name>** is the FQDN of the node.
- ✱ **<namespace>** is the namespace of the target pod.
- ✱ **<pod>** is the name of the target pod.

For example:

```
/proxy/minions/node123.openshift.com/portForward/myns/mypod
```

After sending a port forward request to the API server, the client upgrades the connection to one that supports multiplexed streams; the current implementation uses **SPDY**.

The client creates a stream with the **port** header containing the target port in the pod. All data written to the stream is delivered via the Kubelet to the target pod and port. Similarly, all data sent from the pod for that forwarded connection is delivered back to the same stream in the client.

The client closes all streams, the upgraded connection, and the underlying connection when it is finished with the port forwarding request.



Note

Administrators can see the [Architecture](#) guide for more information.

CHAPTER 26. SHARED MEMORY

26.1. OVERVIEW

There are two types of shared memory objects in Linux: System V and POSIX. The containers in a pod share the IPC namespace of the pod infrastructure container and so are able to share the System V shared memory objects. This document describes how they can also share POSIX shared memory objects.

26.2. POSIX SHARED MEMORY

POSIX shared memory requires that a tmpfs be mounted at ***/dev/shm***. The containers in a pod do not share their mount namespaces so we use volumes to provide the same ***/dev/shm*** into each container in a pod. The following example shows how to set up POSIX shared memory between two containers.

shared-memory.yaml

```
---
apiVersion: v1
id: hello-openshift
kind: Pod
metadata:
  name: hello-openshift
  labels:
    name: hello-openshift
spec:
  volumes:
    - name: dshm
      emptyDir:
        medium: Memory
  containers:
    - image: kubernetes/pause
      name: hello-container1
      ports:
        - containerPort: 8080
          hostPort: 6061
      volumeMounts:
        - mountPath: /dev/shm
          name: dshm
    - image: kubernetes/pause
      name: hello-container2
      ports:
        - containerPort: 8081
          hostPort: 6062
      volumeMounts:
        - mountPath: /dev/shm
          name: dshm
```

1

2

3

specifies the tmpfs volume **dshm**.

2

enables POSIX shared memory for **hello-container1** via **dshm**.

3

enables POSIX shared memory for **hello-container2** via **dshm**.

Create the pod using the ***shared-memory.yaml*** file:

```
$ oc create -f shared-memory.yaml
```

CHAPTER 27. APPLICATION HEALTH

27.1. OVERVIEW

In software systems, components can become unhealthy due to transient issues (such as temporary connectivity loss), configuration errors, or problems with external dependencies. OpenShift Enterprise applications have a number of options to detect and handle unhealthy containers.

27.2. CONTAINER HEALTH CHECKS USING PROBES

A probe is a Kubernetes action that periodically performs diagnostics on a running container. Currently, two types of probes exist, each serving a different purpose:

Liveness Probe	A liveness probe checks if the container in which it is configured is still running. If the liveness probe fails, the kubelet kills the container, which will be subjected to its restart policy. Set a liveness check by configuring the template.spec.containers.livenessprobe stanza of a pod configuration.
Readiness Probe	A readiness probe determines if a container is ready to service requests. If the readiness probe fails a container, the endpoints controller ensures the container has its IP address removed from the endpoints of all services. A readiness probe can be used to signal to the endpoints controller that even though a container is running, it should not receive any traffic from a proxy. Set a readiness check by configuring the template.spec.containers.readinessprobe stanza of a pod configuration.

The exact timing of a probe is controlled by two fields, both expressed in units of seconds:

Field	Description
initialDelaySeconds	How long to wait after the container starts to begin the probe.
timeoutSeconds	How long to wait for the probe to finish (default: 1). If this time is exceeded, OpenShift Enterprise considers the probe to have failed.

Both probes can be configured in three ways:

HTTP Checks

The kubelet uses a web hook to determine the healthiness of the container. The check is deemed successful if the HTTP response code is between 200 and 399. The following is an example of a readiness check using the HTTP checks method:

Example 27.1. Readiness HTTP check

```
...
readinessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
...
```

A HTTP check is ideal for applications that return HTTP status codes when completely initialized.

Container Execution Checks

The kubelet executes a command inside the container. Exiting the check with status 0 is considered a success. The following is an example of a liveness check using the container execution method:

Example 27.2. Liveness Container Execution Check

```
...
livenessProbe:
  exec:
    command:
      - cat
      - /tmp/health
  initialDelaySeconds: 15
  timeoutSeconds: 1
...
```

TCP Socket Checks

The kubelet attempts to open a socket to the container. The container is only considered healthy if the check can establish a connection. The following is an example of a liveness check using the TCP socket check method:

Example 27.3. Liveness TCP Socket Check

```
...
livenessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
...
```

I -

A TCP socket check is ideal for applications that do not start listening until initialization is complete.

For more information on health checks, see the [Kubernetes documentation](#).

CHAPTER 28. EVENTS

28.1. OVERVIEW

Events in OpenShift Enterprise are modeled based on events that happen to API objects in an OpenShift Enterprise cluster. Events allow OpenShift Enterprise to record information about real-world events in a resource-agnostic manner. They also allow developers and administrators to consume information about system components in a unified way.

28.2. VIEWING EVENTS WITH THE CLI

You can get a list of events in a given project using the following command:

```
$ oc get events [-n <project>]
```

28.3. VIEWING EVENTS IN THE CONSOLE

You can see events in your project from the web console from the **Browse** → **Events** page. Many other objects, such as pods and deployments, have their own **Events** tab as well, which shows events related to that object.

28.4. COMPREHENSIVE LIST OF EVENTS

This section describes the events of OpenShift Enterprise.

Table 28.1. Configuration Events

Name	Description
FailedValidation	Failed pod configuration validation.

Table 28.2. Container Events

Name	Description
BackOff	Back-off restarting failed the container.
Created	Container created.

Name	Description
Failed	Pull/Create/Start failed.
Killing	Killing the container.
Started	Container started.

Table 28.3. Health Events

Name	Description
Unhealthy	Container is unhealthy.

Table 28.4. Image Events

Name	Description
BackOff	Back off Ctr Start, image pull.
ErrImageNeverPull	The image's NeverPull Policy is violated.
Failed	Failed to pull the image.
InspectFailed	Failed to inspect the image.
Pulled	Successfully pulled the image or the container image is already present on the machine.
Pulling	Pulling the image.

Table 28.5. Image Manager Events

Name	Description
FreeDiskSpaceFailed	Free disk space failed.
InvalidDiskCapacity	Invalid disk capacity.

Table 28.6. Node Events

Name	Description
FailedMount	Volume mount failed.
HostNetworkNotSupported	Host network not supported.
HostPortConflict	Host/port conflict.
InsufficientFreeCPU	Insufficient free CPU.
InsufficientFreeMemory	Insufficient free memory.
KubeletSetupFailed	Kubelet setup failed.
NilShaper	Undefined shaper.
NodeNotReady	Node is not ready.
NodeNotSchedulable	Node is not schedulable.

Name	Description
NodeReady	Node is ready.
NodeSchedulable	Node is schedulable.
NodeSelectorMismatching	Node selector mismatch.
OutOfDisk	Out of disk.
Rebooted	Node rebooted.
Starting	Starting kubelet.

Table 28.7. Pod Worker Events

Name	Description
FailedSync	Pod sync failed.

Table 28.8. System Events

Name	Description
SystemOOM	There is an OOM (out of memory) situation on the cluster.

CHAPTER 29. DOWNWARD API

29.1. OVERVIEW

It is common for containers to consume information about API objects. The downward API is a mechanism that allows containers to do this without coupling to OpenShift Enterprise. Containers can consume information from the downward API [using environment variables](#) or a [volume plug-in](#).

29.2. SELECTING FIELDS

Fields within the pod are selected using the **FieldRef** API type. **FieldRef** has two fields:

Field	Description
fieldPath	The path of the field to select, relative to the pod.
apiVersion	The API version to interpret the fieldPath selector within.

Currently, these are the valid selectors in the v1 API:

Selector	Description
metadata.name	The pod's name. This is supported in both environment variables and volumes.
metadata.namespace	The pod's namespace. This is supported in both environment variables and volumes.
metadata.labels	The pod's labels. This is only supported in volumes and not in environment variables.
metadata.annotations	The pod's annotations. This is only supported in volumes and not in environment variables.
status.podIP	The pod's IP. This is only supported in environment variables and not volumes.

The **apiVersion** field, if not specified, defaults to the API version of the enclosing pod template.

In the future, more information, such as resource limits for pods and information about services, will be available using the downward API.

29.3. USING ENVIRONMENT VARIABLES

One mechanism for consuming the downward API is using a container's environment variables. The **EnvVar** type's **valueFrom** field (of type **EnvVarSource**) is used to specify that the variable's value should come from a **FieldRef** source instead of the literal value specified by the **value** field. In the future, additional sources may be supported; currently the source's **fieldRef** field is used to select a field from the downward API.

Only constant attributes of the pod can be consumed this way, as environment variables cannot be updated once a process is started in a way that allows the process to be notified that the value of a variable has changed. The fields supported using environment variables are:

✎ Pod name

✎ Pod namespace

For example:

1. Create a **pod.json** file:

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
  - name: env-test-container
    image: gcr.io/google_containers/busybox
    command: [ "/bin/sh", "-c", "env" ]
    env:
    - name: MY_POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: MY_POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
  restartPolicy: Never
```

2. Create the pod from the **pod.json** file:

```
$ oc create -f pod.json
```

3. Check the container's logs:

```
$ oc logs -p dapi-env-test-pod
```

You should see **MY_POD_NAME** and **MY_POD_NAMESPACE** in the logs.

29.4. USING THE VOLUME PLUG-IN

Another mechanism for consuming the downward API is using a volume plug-in. The downward API volume plug-in creates a volume with configured fields projected into files. The **metadata** field of the **VolumeSource** API object is used to configure this volume. The plug-in supports the following fields:

- ✎ Pod name
- ✎ Pod namespace
- ✎ Pod annotations
- ✎ Pod labels

Example 29.1. Downward API Volume Plug-in Configuration

```
spec:
  volumes:
    - name: podinfo
      metadata: 1
        items: 2
          - name: "labels" 3
            fieldRef:
              fieldPath: metadata.labels 4
```

1

The **metadata** field of the volume source configures the downward API volume.

2

The **items** field holds a list of fields to project into the volume.

3

The name of the file to project the field into.

4

The selector of the field to project.

For example:

1. Create a **volume-pod.json** file:

■

```
kind: Pod
apiVersion: v1
metadata:
  labels:
    zone: us-east-coast
    cluster: downward-api-test-cluster1
    rack: rack-123
  name: dapi-volume-test-pod
  annotations:
    annotation1: 345
    annotation2: 456
spec:
  containers:
    - name: volume-test-container
      image: gcr.io/google_containers/busybox
      command: ["sh", "-c", "cat /etc/labels /etc/annotations"]
      volumeMounts:
        - name: podinfo
          mountPath: /etc
          readOnly: false
  volumes:
    - name: podinfo
      metadata:
        items:
          - name: "labels"
            fieldRef:
              fieldPath: metadata.labels
          - name: "annotations"
            fieldRef:
              fieldPath: metadata.annotations
      restartPolicy: Never
```

2. Create the pod from the ***volume-pod.json*** file:

```
$ oc create -f volume-pod.json
```

3. Check the container's logs and verify the presence of the configured fields:

```
$ oc logs -p dapi-volume-test-pod
cluster=downward-api-test-cluster1
rack=rack-123
zone=us-east-coast
annotation1=345
annotation2=456
kubernetes.io/config.source=api
```

CHAPTER 30. MANAGING ENVIRONMENT VARIABLES

30.1. SETTING AND UNSETTING ENVIRONMENT VARIABLES

OpenShift Enterprise provides the **oc set env** command to set or unset environment variables for objects that have a pod [template](#), such as replication controllers or deployment configurations. It can also list environment variables in pods or any object that has a pod template. This command can also be used on **BuildConfig** objects.

30.2. LIST ENVIRONMENT VARIABLES

To list environment variables in pods or pod templates:

```
$ oc set env <object-selection> --list [<common-options>]
```

This example lists all environment variables for pod **p1**:

```
$ oc set env pod/p1 --list
```

30.3. SET ENVIRONMENT VARIABLES

To set environment variables in the pod templates:

```
$ oc set env <object-selection> KEY_1=VAL_1 ... KEY_N=VAL_N [<set-env-  
options>] [<common-options>]
```

Set environment options:

Option	Description
-e, --env=<KEY>=<VAL>	Set given key value pairs of environment variables.
--overwrite	Confirm updating existing environment variables.

In the following example, both commands modify environment variable **STORAGE** in the deployment config **registry**. The first adds, with value **/data**. The second updates, with value **/opt**.

```
$ oc set env dc/registry STORAGE=/data
$ oc set env dc/registry --overwrite STORAGE=/opt
```

The following example finds environment variables in the current shell whose names begin with **RAILS_** and adds them to the replication controller **r1** on the server:

```
$ env | grep RAILS_ | oc set env rc/r1 -e -
```

The following example does not modify the replication controller defined in file **rc.json**. Instead, it

writes a YAML object with updated environment **STORAGE=/local** to new file **rc.yaml**.

```
$ oc set env -f rc.json STORAGE=/opt -o yaml > rc.yaml
```

30.3.1. Automatically Added Environment Variables

Table 30.1. Automatically Added Environment Variables

Variable Name
<SVCNAME>_SERVICE_HOST
<SVCNAME>_SERVICE_PORT

Example Usage

The service **KUBERNETES** which exposes TCP port 53 and has been allocated cluster IP address 10.0.0.11 produces the following environment variables:

```
KUBERNETES_SERVICE_PORT=53
MYSQL_DATABASE=root
KUBERNETES_PORT_53_TCP=tcp://10.0.0.11:53
KUBERNETES_SERVICE_HOST=10.0.0.11
```



Note

Use the **oc rsh** command to SSH into your container and run **oc set env** to list all available variables.

30.4. UNSET ENVIRONMENT VARIABLES

To unset environment variables in the pod templates:

```
$ oc set env <object-selection> KEY_1- ... KEY_N- [<common-options>]
```



Important

The trailing hyphen (-, U+2D) is required.

This example removes environment variables **ENV1** and **ENV2** from deployment config **d1**:

```
$ oc set env dc/d1 ENV1- ENV2-
```

This removes environment variable **ENV** from all replication controllers:

```
$ oc set env rc --all ENV-
```

This removes environment variable **ENV** from container **c1** for replication controller **r1**:

```
$ oc set env rc r1 --containers='c1' ENV-
```

CHAPTER 31. JOBS

31.1. OVERVIEW

A job, in contrast to [a replication controller](#), runs a pod with any number of replicas to completion. A job tracks the overall progress of a task and updates its status with information about active, succeeded, and failed pods. Deleting a job will clean up any pod replicas it created. Jobs are part of the Kubernetes API, which can be managed with **oc** commands like other [object types](#).

See the [Kubernetes documentation](#) for more information about jobs.

31.2. CREATING A JOB

A job configuration consists of the following key parts:

- ✧ A pod template, which describes the application the pod will create.
- ✧ An optional **parallelism** parameter, which specifies how many pod replicas running in parallel should execute a job. If not specified, this defaults to the value in the **completions** parameter.
- ✧ An optional **completions** parameter, specifying how many concurrently running pods should execute a job. If not specified, this value defaults to one.

The following is an example of a **job** resource:

```
apiVersion: extensions/v1beta1
kind: Job
metadata:
  name: pi
spec:
  selector: 1
    matchLabels:
      app: pi
  parallelism: 1 2
  completions: 1 3
  template: 4
    metadata:
      name: pi
      labels:
        app: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
        restartPolicy: Never
```

1. Label selector of the pod to run. It uses the [generalized label selectors](#).
2. Optional value for how many pod replicas a job should run in parallel; defaults to **completions**.

3. Optional value for how many successful pod completions are needed to mark a job completed; defaults to one.
4. Template for the pod the controller creates.

31.3. SCALING A JOB

A job can be scaled up or down by using the **oc scale** command with the **--replicas** option, which, in the case of jobs, modifies the **spec.parallelism** parameter. This will result in modifying the number of pod replicas running in parallel, executing a job.

The following command uses the example job above, and sets the **parallelism** parameter to three:

```
$ oc scale job pi --replicas=3
```



Note

Scaling replication controllers also uses the **oc scale** command with the **--replicas** option, but instead changes the **replicas** parameter of a replication controller configuration.

31.4. SETTING MAXIMUM DURATION

When defining a **Job**, you can define its maximum duration by setting the **activeDeadlineSeconds** field. It is specified in seconds and is not set by default. When not set, there is no maximum duration enforced.

The maximum duration is counted from the time when a first pod gets scheduled in the system, and defines how long a job can be active. It tracks overall time of an execution and is irrelevant to the number of completions (number of pod replicas needed to execute a task). After reaching the specified timeout, the job is terminated by OpenShift Enterprise.

The following example shows the part of a **Job** specifying **activeDeadlineSeconds** field for 30 minutes:

```
spec:
  activeDeadlineSeconds: 1800
```

CHAPTER 32. REVISION HISTORY: DEVELOPER GUIDE

32.1. MON APR 03 2017

Affected Topic	Description of Change
Quotas and Limit Ranges	Removed "m" as a valid suffix for memory in the Compute Resources section.

32.2. TUE MAR 14 2017

Affected Topic	Description of Change
Secrets	Added an example YAML file of a secret that will create four files.

32.3. TUE FEB 21 2017

Affected Topic	Description of Change
Secrets	Corrected an example YAML file and added missing steps.

32.4. MON JAN 30 2017

Affected Topic	Description of Change
Managing Environment Variables	Removed redundant information and CLI reference material; rearranged sections to match user process.
Builds	Updated the example Dockerfile path to point to a file, not a directory.

32.5. MON JAN 16 2017

Affected Topic	Description of Change
Managing Images	Added information about the supports annotation on image streams.

32.6. MON JAN 09 2017

Affected Topic	Description of Change
Templates	Updated the oc export all command example.
Builds	Added the FROM Image section.

32.7. TUE OCT 11 2016

Affected Topic	Description of Change
Copying Files to or from a Container	Added a procedure outlining how oc rsync can be used to copy database archives from an existing database container to a new database container's persistent volume directory.

32.8. TUE OCT 04 2016

Affected Topic	Description of Change
Builds	Added information on shallow cloning.

32.9. TUE SEP 13 2016

Affected Topic	Description of Change
Using Daemonsets	New topic on using daemonsets as a developer.

32.10. TUE SEP 06 2016

Affected Topic	Description of Change
Deployments	Added a new Running a Pod with a Different Service Account section.
Migrating Applications → Migrating Database Applications	Fixed the formatting of some commands.
Events	Added a comprehensive list of events .

32.11. MON AUG 29 2016

Affected Topic	Description of Change
Migrating Applications	<p>Added a new set of topics reviewing the migration procedure of OpenShift version 2 (v2) applications to OpenShift version 3 (v3), including:</p> <ul style="list-style-type: none">✦ Migrating Database Applications✦ Migrating Web Framework Applications✦ QuickStart Examples✦ Continuous Integration and Deployment (CI/CD)✦ Webhooks and Action Hooks✦ S2I Tool✦ Support Guide

32.12. MON AUG 08 2016

Affected Topic	Description of Change
Using Persistent Volumes	Added a <code>spec.volumeName</code> field to the Requesting Storage example.

32.13. MON AUG 01 2016

Affected Topic	Description of Change
Integrating External Services	Corrected the endpoints example within the External MySQL Database section.

32.14. WED JUL 27 2016

Affected Topic	Description of Change
Builds	Added Build Resources section.
Downward API	Added support details in the Selecting Fields section.
Application Health	Removed High-level Application Health Checks section.
Creating New Applications	Added the Useful Edits section with instructions on how to deploy an application to selected nodes .

32.15. THU JUL 14 2016

Affected Topic	Description of Change
Builds	Fixed build configuration example in the Other section.
Managing Images	Updated the oc secrets new --help command to be oc secrets new-dockercfg --help .
Deployments	Clarified operational conditions around config-change and image-change triggers.
	Added the mid lifecycle-hook in the Recreate Strategy section.
Secrets	Added clarifying details to the Restrictions section.

Affected Topic	Description of Change
Managing Volumes	Added configmap to the list of supported values for the --type option of the oc volume command.
Port Forwarding	Updated outdated syntax instances of oc port-forward -p .
Downward API	Added status.podIP as a valid selector in the v1 API.
Managing Environment Variables	Added information about automatically added environment variables.

32.16. TUE JUN 14 2016

Affected Topic	Description of Change
Quotas and Limit Ranges	Added a section on project resource limits.

32.17. FRI JUN 10 2016

Affected Topic	Description of Change
Opening a Remote Shell to Containers	Added a new topic on opening a remote shell to containers.

32.18. MON MAY 30 2016

Affected Topic	Description of Change
Templates	Fixed oc process example in the Parameters section.
Copying Files to or from a Container	Added use cases for the oc rsync command to the Overview.

Affected Topic	Description of Change
Builds	Updated the examples in the Defining a BuildConfig , Git Repository Source Options , and Using a Proxy for Git Cloning sections to use https for GitHub access.

32.19. THU MAY 12 2016

OpenShift Enterprise 3.2 initial release.

Affected Topic	Description of Change
Builds	Added information about binary builds to the Binary Source section.
	Clarified how to avoid copying the base directory when including extra files in the image source .
	Added a Troubleshooting Guidance table.
	Added a Using Secrets During a Build section.
	Added a Build Hooks section.
	Added an Image Source section.
	Added a Deleting a BuildConfig section.
Jobs	Added a Setting Maximum Duration section, which includes job deadline information.
Resource Quota	Moved the topic from Developer Guide to Cluster Administration, as it involves cluster administration tasks, and renamed it to Setting Quotas .
ConfigMaps	New topic for the new ConfigMap object.

Affected Topic	Description of Change
Managing Images	<p>New topic aggregating many related tasks regarding images and image streams. Includes many sections previously found in the Builds and Image Streams and Image Pull Secrets topics, as well as updated and enhanced details throughout.</p> <hr/> <p>Added an Important box to Adding Tags to Image Streams advising against tagging internally managed images.</p> <hr/> <p>Added a Creating an Image Stream by Manually Pushing an Image section.</p> <hr/> <p>Added an Importing Images from Private Registries section.</p>
Quotas and Limit Ranges	Consolidated and re-used developer-relevant information about quotas and limit ranges from related Cluster Administrator topics into what was previously the "Compute Resources" topic, and renamed it to Quotas and Limit Ranges .
Service Accounts	Updated to use the oc create serviceaccount command.
Application Life Cycle Examples	Added images to the Application Life Cycle Examples topic.
Managing Volumes	Added a Note indicating that EmptyDir volume storage may be restricted by a quota based on the pods FSGroup, if enabled by your cluster administrator.
Application Life Cycle Examples	Added the Application Life Cycle Examples topic to the Developer Guide, which outlines example workflows for building applications.
Projects	Added a Note box about project creation limits.
Pod Autoscaling	Updated to include oc autoscale usage.