



# **OpenShift Enterprise 3.0**

## **Developer Guide**

OpenShift Enterprise 3.0 Developer Reference



# OpenShift Enterprise 3.0 Developer Guide

---

OpenShift Enterprise 3.0 Developer Reference

## Legal Notice

Copyright © 2018 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

<b>CHAPTER 1. OVERVIEW</b>	<b>7</b>
<b>CHAPTER 2. AUTHENTICATION</b>	<b>8</b>
2.1. WEB CONSOLE AUTHENTICATION	8
2.2. CLI AUTHENTICATION	8
<b>CHAPTER 3. PROJECTS</b>	<b>10</b>
3.1. OVERVIEW	10
3.2. CREATING A PROJECT	10
3.3. VIEWING PROJECTS	10
3.4. CHECKING PROJECT STATUS	11
3.5. FILTERING BY LABELS	11
3.6. DELETING A PROJECT	13
<b>CHAPTER 4. SERVICE ACCOUNTS</b>	<b>14</b>
4.1. OVERVIEW	14
4.2. USERNAMES AND GROUPS	14
4.3. DEFAULT SERVICE ACCOUNTS AND ROLES	14
4.4. MANAGING SERVICE ACCOUNTS	15
4.5. MANAGING SERVICE ACCOUNT CREDENTIALS	15
4.6. MANAGING ALLOWED SECRETS	15
4.7. USING A SERVICE ACCOUNT'S CREDENTIALS INSIDE A CONTAINER	16
4.8. USING A SERVICE ACCOUNT'S CREDENTIALS EXTERNALLY	17
<b>CHAPTER 5. CREATING NEW APPLICATIONS</b>	<b>18</b>
5.1. OVERVIEW	18
5.2. USING THE CLI	18
5.2.1. Specifying Source Code	18
5.2.1.1. Build Strategy Detection	19
5.2.1.2. Language Detection	19
5.2.2. Specifying an Image	20
5.2.3. Specifying a Template	21
5.2.3.1. Template Parameters	21
5.2.4. Specifying Environment Variables	21
5.2.5. Specifying Labels	21
5.2.6. Command Output	22
5.2.6.1. Output Without Creation	22
5.2.6.2. Object names	22
5.2.6.3. Object Project or Namespace	22
5.2.6.4. Artifacts Created	23
5.2.7. Advanced: Multiple Components and Grouping	23
5.2.7.1. Grouping Images and Source in a Single Pod	23
5.3. USING THE WEB CONSOLE	24
<b>CHAPTER 6. TEMPLATES</b>	<b>28</b>
6.1. OVERVIEW	28
6.2. UPLOADING A TEMPLATE	28
6.3. CREATING FROM TEMPLATES USING THE WEB CONSOLE	28
6.4. CREATING FROM TEMPLATES USING THE CLI	30
6.4.1. Labels	30
6.4.2. Parameters	30
6.4.3. Generating a List of Objects	31
6.5. MODIFYING AN UPLOADED TEMPLATE	32

6.6. USING THE INSTANTAPP TEMPLATES	32
6.7. WRITING TEMPLATES	33
6.7.1. Description	33
6.7.2. Labels	33
6.7.3. Parameters	34
6.7.4. Object List	35
6.7.5. Creating a Template from Existing Objects	36
<b>CHAPTER 7. BUILDS</b> .....	<b>37</b>
7.1. OVERVIEW	37
7.2. DEFINING A BUILDCONFIG	37
7.3. SOURCE-TO-IMAGE STRATEGY OPTIONS	38
7.3.1. Force Pull	38
7.3.2. Incremental Builds	39
7.3.3. Override Builder Image Scripts	39
7.3.4. Environment Variables	40
7.3.4.1. Environment Files	40
7.3.4.2. BuildConfig Environment	40
7.4. DOCKER STRATEGY OPTIONS	41
7.4.1. No Cache	41
7.4.2. Force Pull	41
7.4.3. Environment Variables	41
7.5. CUSTOM STRATEGY OPTIONS	42
7.5.1. Exposing the Docker Socket	42
7.5.2. Secrets	42
7.5.3. Force Pull	43
7.5.4. Environment Variables	43
7.6. USING A PROXY FOR GIT CLONING	44
7.6.1. Using Private Repositories for Builds	44
7.7. DOCKERFILE SOURCE	45
7.8. BINARY SOURCE	45
7.9. IMAGE SOURCE	46
7.10. USING SECRETS DURING A BUILD	47
7.10.1. Defining Secrets in the BuildConfig	47
7.10.2. Source-to-Image Strategy	48
7.10.3. Docker Strategy	49
7.10.4. Custom Strategy	49
7.11. STARTING A BUILD	49
7.12. CANCELING A BUILD	49
7.13. ACCESSING BUILD LOGS	49
7.14. SOURCE CODE	50
7.15. BUILD TRIGGERS	51
7.15.1. Webhook Triggers	51
7.15.2. Image Change Triggers	52
7.15.3. Configuration Change Triggers	54
7.16. USING DOCKER CREDENTIALS FOR PUSHING AND PULLING IMAGES	55
<b>CHAPTER 8. DEPLOYMENTS</b> .....	<b>57</b>
8.1. OVERVIEW	57
8.2. CREATING A DEPLOYMENT CONFIGURATION	57
8.3. STARTING A DEPLOYMENT	58
8.4. VIEWING A DEPLOYMENT	59
8.5. CANCELING A DEPLOYMENT	59

8.6. RETRYING A DEPLOYMENT	59
8.7. ROLLING BACK A DEPLOYMENT	60
8.8. TRIGGERS	60
8.8.1. Configuration Change Trigger	60
8.8.2. Image Change Trigger	61
8.9. STRATEGIES	61
8.9.1. Rolling Strategy	62
8.9.2. Recreate Strategy	63
8.9.3. Custom Strategy	64
8.10. LIFECYCLE HOOKS	64
8.10.1. Pod-based Lifecycle Hook	65
8.11. DEPLOYMENT RESOURCES	66
8.12. MANUAL SCALING	67
<b>CHAPTER 9. ROUTES</b>	<b>68</b>
9.1. OVERVIEW	68
9.2. CREATING ROUTES	68
<b>CHAPTER 10. INTEGRATING EXTERNAL SERVICES</b>	<b>70</b>
10.1. OVERVIEW	70
10.2. EXTERNAL MYSQL DATABASE	70
10.3. EXTERNAL SAAS PROVIDER	72
<b>CHAPTER 11. SECRETS</b>	<b>76</b>
11.1. OVERVIEW	76
11.2. PROPERTIES OF SECRETS	76
11.2.1. Secrets and the Pod Lifecycle	76
11.3. CREATING AND USING SECRETS	77
11.3.1. Creating Secrets	77
11.3.2. Secrets in Volumes	77
11.3.3. Image Pull Secrets	77
11.4. RESTRICTIONS	77
11.4.1. Secret Data Keys	77
11.5. EXAMPLES	78
<b>CHAPTER 12. IMAGE PULL SECRETS</b>	<b>79</b>
12.1. OVERVIEW	79
12.2. INTEGRATED REGISTRY AUTHENTICATION AND AUTHORIZATION	79
12.2.1. Allowing Pods to Reference Images Across Projects	79
12.3. ALLOWING PODS TO REFERENCE IMAGES FROM OTHER SECURED REGISTRIES	79
<b>CHAPTER 13. RESOURCE LIMITS</b>	<b>81</b>
13.1. OVERVIEW	81
13.2. LIMITS	81
13.3. LIMIT ENFORCEMENT	81
13.4. CREATING A LIMIT RANGE	83
13.5. VIEWING LIMITS	83
13.6. DELETING LIMITS	83
<b>CHAPTER 14. RESOURCE QUOTA</b>	<b>84</b>
14.1. OVERVIEW	84
14.2. USAGE LIMITS	84
14.3. QUOTA ENFORCEMENT	84
14.4. SAMPLE RESOURCE QUOTA FILE	85
14.5. CREATE A QUOTA	85

14.6. VIEW A QUOTA	85
14.7. CONFIGURING THE QUOTA SYNCHRONIZATION PERIOD	86
<b>CHAPTER 15. MANAGING VOLUMES</b>	<b>87</b>
15.1. OVERVIEW	87
15.2. GENERAL CLI USAGE	87
15.3. ADDING VOLUMES	88
15.4. UPDATING VOLUMES	89
15.5. REMOVING VOLUMES	89
15.6. LISTING VOLUMES	90
<b>CHAPTER 16. USING PERSISTENT VOLUMES</b>	<b>92</b>
16.1. OVERVIEW	92
16.2. REQUESTING STORAGE	92
16.3. VOLUME AND CLAIM BINDING	92
16.4. CLAIMS AS VOLUMES IN PODS	93
<b>CHAPTER 17. EXECUTING REMOTE COMMANDS</b>	<b>94</b>
17.1. OVERVIEW	94
17.2. BASIC USAGE	94
17.3. PROTOCOL	94
<b>CHAPTER 18. PORT FORWARDING</b>	<b>96</b>
18.1. OVERVIEW	96
18.2. BASIC USAGE	96
18.3. PROTOCOL	96
<b>CHAPTER 19. SHARED MEMORY</b>	<b>98</b>
19.1. OVERVIEW	98
19.2. POSIX SHARED MEMORY	98
<b>CHAPTER 20. APPLICATION HEALTH</b>	<b>100</b>
20.1. OVERVIEW	100
20.2. CONTAINER HEALTH CHECKS USING PROBES	100
<b>CHAPTER 21. EVENTS</b>	<b>102</b>
21.1. OVERVIEW	102
21.2. QUERYING EVENTS WITH THE CLI	102
21.3. VIEWING EVENTS IN THE CONSOLE	102
21.4. OUT-OF-BAND INTERFACE	102
<b>CHAPTER 22. DOWNWARD API</b>	<b>103</b>
22.1. OVERVIEW	103
22.2. SELECTING FIELDS	103
22.3. USING ENVIRONMENT VARIABLES	103
22.4. USING THE VOLUME PLUG-IN	104
<b>CHAPTER 23. MANAGING ENVIRONMENT VARIABLES</b>	<b>107</b>
23.1. OVERVIEW	107
23.2. CLI INTERFACE	107
23.3. SET ENVIRONMENT VARIABLES	108
23.4. UNSET ENVIRONMENT VARIABLES	108
23.5. LIST ENVIRONMENT VARIABLES	109
<b>CHAPTER 24. REVISION HISTORY: DEVELOPER GUIDE</b>	<b>110</b>
24.1. WED MAY 25 2016	110



24.2. THU MAY 19 2016	110
24.3. TUE APR 19 2016	110
24.4. THU MAR 17 2016	110
24.5. MON FEB 15 2016	110
24.6. TUE JUN 23 2015	110



# CHAPTER 1. OVERVIEW

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

1. Monitor and browse projects with the web console
2. Configure and utilize the CLI
3. Generate configurations using templates
4. Manage builds and webhooks
5. Define and trigger deployments
6. Integrate external services (databases, SaaS endpoints)

## CHAPTER 2. AUTHENTICATION

### 2.1. 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](#).

### 2.2. 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 server with the provided credentials. If any information required to successfully log in to an OpenShift 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]
```

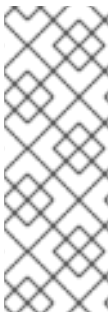
The following table describes these common options:

**Table 2.1. Common CLI Configuration Options**

Option	Syntax	Description
<b>-s, --server</b>	<code>\$ oc login -s=&lt;server&gt;</code>	Specifies the host name of the OpenShift 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.
<b>-u, --username and -p, --password</b>	<code>\$ oc login -u=&lt;username&gt; -p=&lt;password&gt;</code>	Allows you to specify the credentials to log in to the OpenShift 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.

Option	Syntax	Description
<b>-n, --namespace</b>	<pre>\$ oc login -u= &lt;username&gt; -p= &lt;password&gt; -n= &lt;project&gt;</pre>	A global CLI option which, when used with <b>oc login</b> , allows you to specify the project to switch to when logging in as a given user.
<b>--certificate-authority</b>	<pre>\$ oc login -- certificate- authority= &lt;path/to/file. crt&gt;</pre>	Correctly and securely authenticates with an OpenShift server that uses HTTPS. The path to a certificate authority file must be provided.
<b>--insecure-skip-tls-verify</b>	<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 <b>oc login</b> to a HTTPS server that does not provide a valid certificate, and this or the <b>--certificate-authority</b> flags were not provided, <b>oc login</b> 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 3. PROJECTS

### 3.1. OVERVIEW

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

### 3.2. 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"
```

### 3.3. 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.

### 3.4. 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:

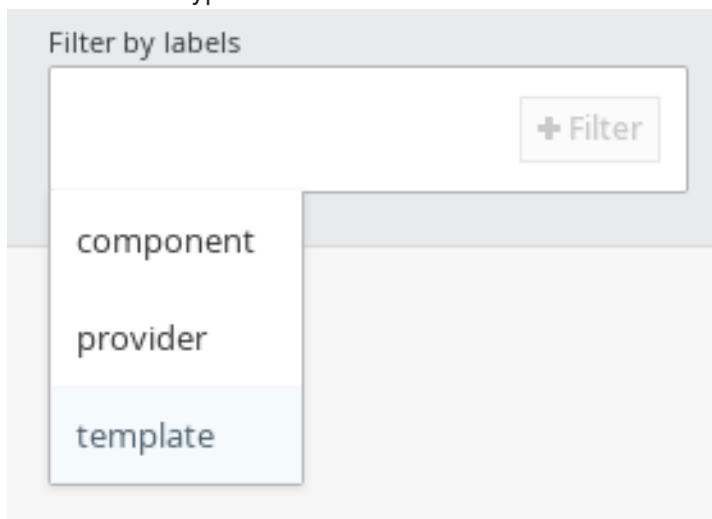
```
$ oc status
```

### 3.5. 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:

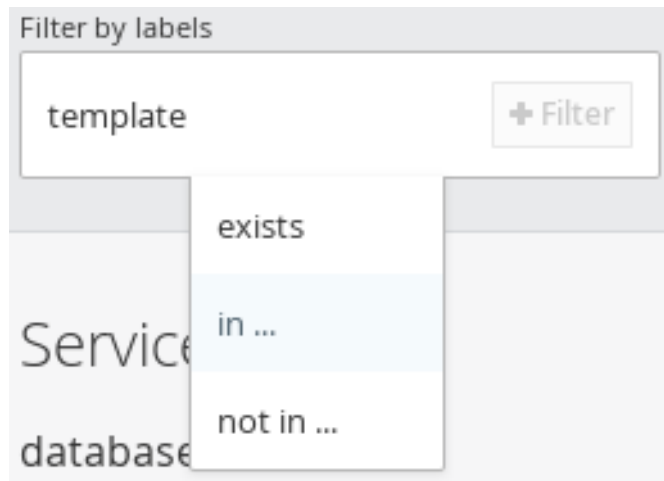


2. Select one of the following:

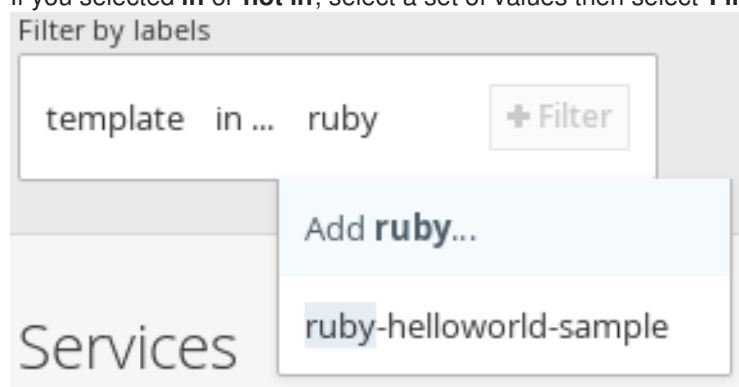
exists	Verify that the label name exists, but ignore its value.
in	Verify that the label name exists and is equal to one of the selected values.

not in

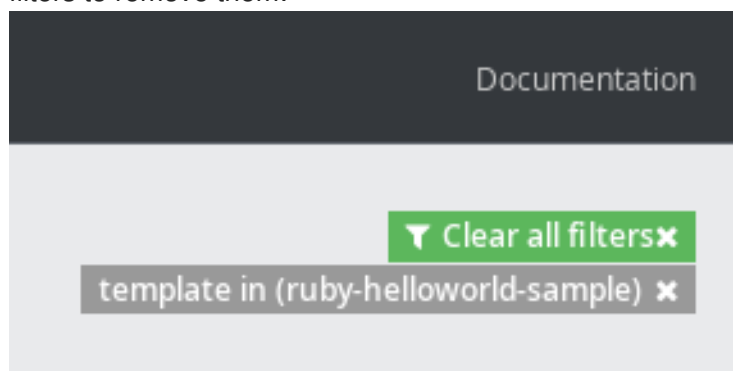
Verify that the label name does not exist, or is not equal to any of the selected values.



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



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





## 3.6. 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.

To delete a project:

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

## CHAPTER 4. SERVICE ACCOUNTS

### 4.1. OVERVIEW

When a person uses the command line 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.

### 4.2. USERNAMES AND GROUPS

Every service account has an associated username that can be granted roles, just like a regular user. The username 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**, which includes all service accounts in the system
- **system:serviceaccounts:<project>**, which 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
```

### 4.3. DEFAULT SERVICE ACCOUNTS AND ROLES

Three service accounts are automatically created in every project:

- **builder** is 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** is used by deployment pods and is given the **system:deployer** role, which allows viewing and modifying replication controllers and pods in the project.
- **default** is 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.

## 4.4. MANAGING SERVICE ACCOUNTS

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

```
$ more sa.json
{
  "apiVersion": "v1",
  "kind": "ServiceAccount",
  "metadata": {
    "name": "robot"
  }
}

$ oc create -f sa.json
serviceaccounts/robot
```

## 4.5. 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.

## 4.6. 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
```

## 4.7. 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"
  ]
}
```

## 4.8. USING A SERVICE ACCOUNT'S CREDENTIALS EXTERNALLY

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

Use **oc describe secret <secret-name>** to view a service account's API token:

```
$ 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 5. CREATING NEW APPLICATIONS

### 5.1. OVERVIEW

You can create a new OpenShift application using the [web console](#) or by running the `oc new-app` command [from the CLI](#). OpenShift creates a new application by specifying [source code](#), [images](#), or [templates](#). The `new-app` command looks for images on the local Docker installation (if available), in a [Docker registry](#), or an OpenShift [image stream](#).

If you specify source code, `new-app` attempts to construct a [build configuration](#) that builds your source into a new application [image](#). It also constructs a [deployment configuration](#) that deploys that new image, and a [service](#) to provide load balanced access to the deployment that is running your image.



#### NOTE

If you [specify source code](#), you may need to [run a build](#) with `oc start-build` after the application is created.

### 5.2. USING THE CLI

You can create a new application using the `oc new-app` command from [the CLI](#).

#### 5.2.1. Specifying Source Code

The `new-app` command allows you to create applications using source code from a local or remote Git repository. If only a source repository is specified, `new-app` tries to automatically [determine](#) the type of [build strategy](#) to use (**Docker** or **Source**), and in the case of **Source** type builds, [an appropriate language builder image](#).

You can tell `new-app` to use a subdirectory of your source code repository by specifying a `--context-dir` flag. Also, when specifying a remote URL, you can specify a Git reference to use by appending `# [reference]` to the end of the URL.



#### NOTE

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

#### Example 5.1. To Create an Application Using the Git Repository at the Current Directory:

```
$ oc new-app .
```

#### Example 5.2. 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
```

**Example 5.3. To Create an Application Using a Remote Git Repository with a Specific Branch Reference:**

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

### 5.2.1.1. Build Strategy Detection

If **new-app** finds a **Dockerfile** in the repository, it generates a **Docker build strategy**. Otherwise, it generates a **Source strategy**. To use a specific strategy, set the **--strategy** flag to either **source** or **docker**.

**Example 5.4. To Force new-app to Use the Docker Strategy for a Local Source Repository:**

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

### 5.2.1.2. Language Detection

If creating a **Source** build, **new-app** attempts to determine which language builder to use based on 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, config.py</i>
perl	<i>index.pl, cpanfile</i>

After a language is detected, **new-app** searches the OpenShift 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.

To override the image that **new-app** uses as the builder for a particular source repository, the image (either an image stream or Docker specification) can be specified along with the repository using a **~** as a separator.

**Example 5.5. To Use Image Stream myproject/my-ruby to Build the Source at a Remote GitHub Repository:**

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

**Example 5.6. To Use Docker Image `openshift/ruby-20-centos7:latest` to Build Source in a Local Repository:**

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

### 5.2.2. Specifying an Image

The **new-app** command generates the necessary artifacts to deploy an existing image as an application. Images can come from image streams in the OpenShift server, images in a specific registry or [Docker Hub](#), 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 cluster nodes.

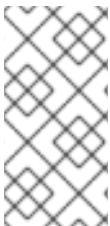
**Example 5.7. To Create an Application from the [DockerHub MySQL Image](#):**

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

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

**Example 5.8. To Create an Application from a Local Registry:**

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



#### NOTE

If the registry that the image comes from is not [secured with SSL](#), cluster administrators must ensure that the Docker daemon on the OpenShift nodes 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.

To create an application from an existing [image stream](#), specify the namespace (optional), name, and tag (optional) for the image stream.

**Example 5.9. To Create an Application from an Existing Image Stream with a Specific Tag:**

```
-
```



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

### 5.2.3. Specifying a Template

The **new-app** command can instantiate a [template](#) from a previously stored template or from a template file. To instantiate a previously stored template, specify the name of the template as an argument. For example, store a [sample application template](#) and use it to create an application.

#### Example 5.10. To Create an Application from a Previously Stored Template:

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

To use a template in the file system directly, without first storing it in OpenShift, use the **-f** | **--file** argument or simply specify the file name as the argument to **new-app**.

#### Example 5.11. To Create an Application from a Template in a File:

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

#### 5.2.3.1. Template Parameters

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

#### Example 5.12. To Specify Template Parameters with a Template:

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

### 5.2.4. Specifying Environment Variables

When generating applications from [source](#) or an [image](#), you can use the **-e** | **--env** argument to specify environment to be passed to the application container at run time.

#### Example 5.13. To Set Environment Variables When Creating an Application for a Database Image:

```
$ oc new-app openshift/postgresql-92-centos7 \
  -e POSTGRES_USER=user \
  -e POSTGRES_DATABASE=db \
  -e POSTGRES_PASSWORD=password
```

### 5.2.5. Specifying Labels

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

**Example 5.14. To Use the Label Argument to Label Objects Created by new-app:**

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

## 5.2.6. Command Output

The **new-app** command generates OpenShift 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.

### 5.2.6.1. Output Without Creation

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

**Example 5.15. To Output new-app Artifacts to a File, Edit Them, Then Create Them Using oc create:**

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

### 5.2.6.2. Object 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.

**Example 5.16. To Create new-app Artifacts with a Different Name:**

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

### 5.2.6.3. Object Project or Namespace

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

**Example 5.17. To Create new-app Artifacts in a Different Project:**

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

#### 5.2.6.4. Artifacts Created

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

**Table 5.2. new-app Output Objects**

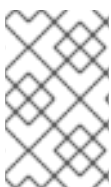
Artifact	Description
<b>BuildConfig</b>	A <b>BuildConfig</b> is created for each source repository specified in the command line. The <b>BuildConfig</b> specifies the strategy to use, the source location, and the build output location.
<b>ImageStreams</b>	For <b>BuildConfig</b> , two <b>ImageStreams</b> are usually created: one to represent the input image (the builder image in the case of <b>Source</b> builds or <b>FROM</b> image in case of <b>Docker</b> builds), and another one to represent the output image. If a Docker image was specified as input to <b>new-app</b> , then an image stream is created for that image as well.
<b>DeploymentConfig</b>	A <b>DeploymentConfig</b> is created either to deploy the output of a build, or a specified image. The <b>new-app</b> command creates <a href="#">EmptyDir volumes</a> for all Docker volumes that are specified in containers included in the resulting <b>DeploymentConfig</b> .
<b>Service</b>	The <b>new-app</b> 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 <b>new-app</b> has completed, simply use the <b>oc expose</b> command to generate additional services.
Other	Other objects can be generated when instantiating <a href="#">templates</a> .

#### 5.2.7. Advanced: Multiple Components and Grouping

The **new-app** command allows creating multiple applications from [source](#), [images](#), or [templates](#) at once. To do this, simply specify multiple parameters to the **new-app** call. Labels specified in the command line apply to all objects created by the single call. Environment variables apply to all components created from source or images.

##### Example 5.18. 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.

##### 5.2.7.1. 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 which images should be grouped together. To group the image built from a source repository with other images, specify its builder image in the group.

**Example 5.19. To Deploy Two Images in a Single Pod:**

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

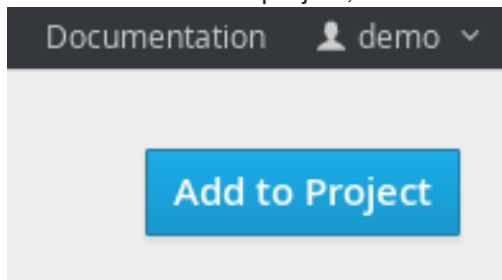
**Example 5.20. To Deploy an Image Built from Source and an External Image Together:**

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

## 5.3. USING THE WEB CONSOLE

You can also create applications using the [web console](#):

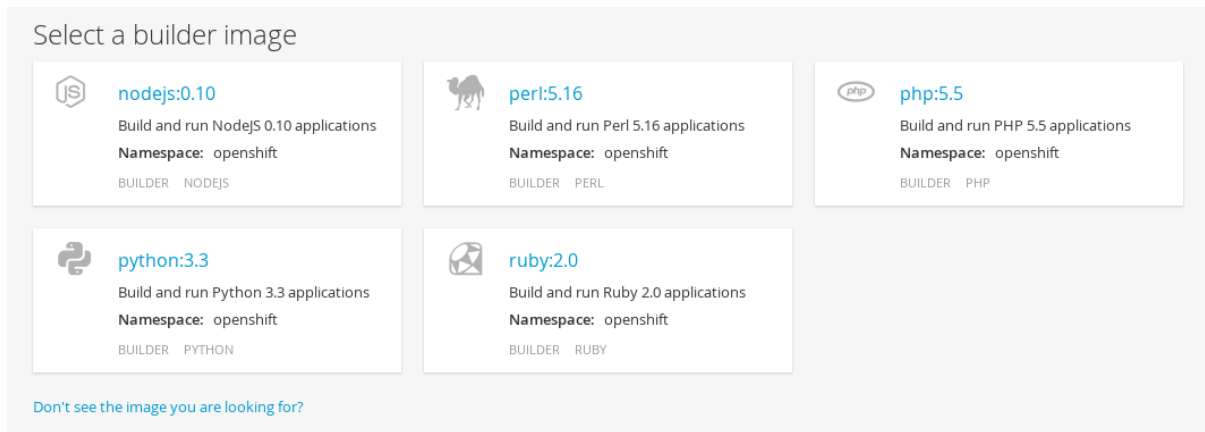
1. While in the desired project, click **Add to Project**:



2. Enter the repository URL for the application to build:

A screenshot of the 'Create Using Your Code' form in the OpenShift web console. The form has a light gray background. At the top, the title 'Create Using Your Code' is displayed in a large, dark font. Below the title, a subtitle reads 'Create your application from a Git source code repository.' There is a text input field with the placeholder text 'Repository URL'. To the right of the input field is a button labeled 'Next'. Below the input field, there is a small text line that says 'For example, <https://github.com/openshift/nodejs-ex>'.

3. Select either a builder image from the list of images in your project, or from the global library:




## NOTE

Only [image stream tags](#) which have the **builder** tag listed in their annotations will 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** will appear in the web console as a builder.

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



# nodejs <sup>1</sup>

Version 0.10  
Build and run NodeJS 0.10 applications

## Name <sup>2</sup>

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

*Note:* After creation, these settings can only be modified through the oc command.

## Routing <sup>3</sup>

[Edit Routing](#) [About Routing](#)

Create a route to the application: Yes

## Deployment Configuration <sup>4</sup>

[Edit Deployment Configuration](#) [About Deployment Configuration](#)

Autodeploy when

New image is available: Yes

Deployment configuration changes: Yes

Environment Variables [?](#)

None

## Build Configuration <sup>5</sup>

[Edit Build Configuration](#) [About Build Configuration](#)

Source Code: <https://github.com/openshift/nodejs-ex>

Automatically build new image when code changes: Yes

Automatically build new image when builder image changes: Yes

## Scaling <sup>6</sup>

[Edit Scaling](#) [About Scaling](#)

Replicas: 1

## Labels <sup>7</sup>

[Edit labels](#) [About labels](#)

None

The builder image name and description.

The application name used for the generated OpenShift objects.

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.

## CHAPTER 6. TEMPLATES

### 6.1. OVERVIEW

A [template](#) describes a set of [objects](#) that can be parameterized and processed to produce a list of objects for creation by OpenShift. 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](#).

### 6.2. UPLOADING A TEMPLATE

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

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:

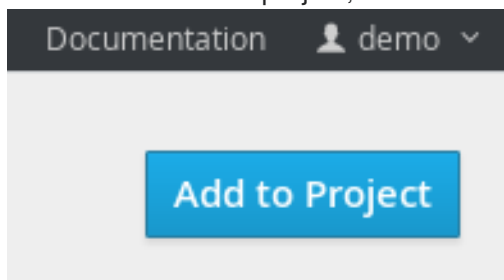
```
$ oc create -f <filename> -n <project>
```

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

### 6.3. 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:





## Create Using a Template


Templates have predefined resources for quickly creating components. You can customize some options in the next step.


### Instant Apps



**cakephp-example**  
 An example CakePHP application with no database  
**Namespace:** openshift  
 INSTANT-APP PHP CAKEPHP


**cakephp-mysql-example**  
 An example CakePHP application with a MySQL database  
**Namespace:** openshift  
 INSTANT-APP PHP CAKEPHP MYSQL



**dancer-example**  
 An example Dancer application with no database  
**Namespace:** openshift  
 INSTANT-APP PERL DANCER


**dancer-mysql-example**  
 An example Dancer application with a MySQL database


**django-example**  
 An example Django application with no database


**django-postgresql-example**  
 An example Django application with a PostgreSQL database

### 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](#)

<b>SOURCE_REPOSITORY_URL</b>	https://github.com/openshift/rails-ex.git
The URL of the repository with your application source code	
<b>SOURCE_REPOSITORY_REF</b>	
Set this to a branch name, tag or other ref of your repository if you are not using the default branch	
<b>CONTEXT_DIR</b>	
Set this to the relative path to your project if it is not in the root of your repository	
<b>APPLICATION_DOMAIN</b>	rails-example.openshiftapps.com
The exposed hostname that will route to the Rails service	
<b>GITHUB_WEBHOOK_SECRET</b>	(generated)
A secret string used to configure the GitHub webhook	
<b>SECRET_KEY_BASE</b>	(generated)
Your secret key for verifying the integrity of signed cookies	
<b>APPLICATION_USER</b>	openshift
The application user that is used within the sample application to authorize access on pages	
<b>APPLICATION_PASSWORD</b>	secret
The application password that is used within the sample application to authorize access on pages	
<b>DATABASE_SERVICE_NAME</b>	postgresql
Database service name	
<b>POSTGRESQL_USER</b>	(generated)
database username	
<b>POSTGRESQL_PASSWORD</b>	(generated)
database password	
<b>POSTGRESQL_DATABASE</b>	root
database name	
<b>POSTGRESQL_MAX_CONNECTIONS</b>	10
database max connections	
<b>POSTGRESQL_SHARED_BUFFERS</b>	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 for objects.

## 6.4. CREATING FROM TEMPLATES USING THE CLI

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

### 6.4.1. 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
```

### 6.4.2. 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 InstantApp 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.

### 6.4.3. Generating a List of Objects

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

```
$ 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 6.1. 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 -
```

## 6.5. 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>
```

## 6.6. USING THE INSTANTAPP TEMPLATES

OpenShift provides a number of default InstantApp 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 InstantApp templates with:

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

If they are not available, direct your cluster administrator to the [First Steps](#) 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 InstantApp templates define a database [deployment configuration](#). The configuration they define uses ephemeral storage for the database content. These templates should be used for demonstration purposes only as all database data will be lost if the database pod restarts for any reason.

## 6.7. 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.

### 6.7.1. 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 InstantApps 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.

### 6.7.2. 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.

### 6.7.3. 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",
  ...
  {
    "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.

#### 6.7.4. 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.

### 6.7.5. 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>
```

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



## CHAPTER 7. BUILDS

### 7.1. OVERVIEW

A **build** is a process of creating runnable images to be used on OpenShift. There are three build strategies:

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

### 7.2. 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 Docker image tag or the source code changes:

#### Example 7.1. BuildConfig Object Definition

```
{
  "kind": "BuildConfig",
  "apiVersion": "v1",
  "metadata": {
    "name": "ruby-sample-build" ❶
  },
  "spec": {
    "triggers": [ ❷
      {
        "type": "GitHub",
        "github": {
          "secret": "secret101"
        }
      },
      {
        "type": "Generic",
        "generic": {
          "secret": "secret101"
        }
      },
      {
        "type": "ImageChange"
      }
    ],
    "source": { ❸
      "type": "Git",
      "git": {
        "uri": "https://github.com/openshift/ruby-hello-world"
      }
    }
  },
}
```

```

    "strategy": { ❹
      "type": "Source",
      "sourceStrategy": {
        "from": {
          "kind": "ImageStreamTag",
          "name": "ruby-20-centos7:latest"
        }
      }
    },
    "output": { ❺
      "to": {
        "kind": "ImageStreamTag",
        "name": "origin-ruby-sample:latest"
      }
    }
  }
}

```

- ❶ This specification will create a new **BuildConfig** named **ruby-sample-build**.
- ❷ You can specify a list of [triggers](#), which cause a new build to be created.
- ❸ The **source** section defines the source code repository location. You can provide additional options, such as **sourceSecret** or **contextDir** here.
- ❹ 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** Docker image that Source-To-Image will use for the application build.
- ❺ After the Docker image is successfully built, it will be pushed into the repository described in the **output** section.

## 7.3. SOURCE-TO-IMAGE STRATEGY OPTIONS

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

### 7.3.1. 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" ❶
      }
    },
    "forcePull": true ❷
  }
}

```

```

    }
  }
}

```

- 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.

### 7.3.2. 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. The S2I images provided by OpenShift do not implement artifact reuse, so setting **incremental** to **true** will have no effect on builds using those builder images.
- 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.

### 7.3.3. Override 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 **.sti/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 **.sti/bin** of the source repository. See the [S2I Requirements](#) topic and the [S2I documentation](#) for information on how S2I scripts are used.

### 7.3.4. 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.

#### 7.3.4.1. Environment Files

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

If you provide a **.sti/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 **.sti/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 **.sti/environment** file to cause the Rails application to start in **development** mode instead of **production**.

#### 7.3.4.2. 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"
      }
    ]
  }
}
```

## 7.4. DOCKER STRATEGY OPTIONS

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

### 7.4.1. 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
```

### 7.4.2. 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
    }
  }
}
```

- 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.

### 7.4.3. 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/"
      }
    ]
  }
}
```

## 7.5. CUSTOM STRATEGY OPTIONS

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

### 7.5.1. Exposing the Docker Socket

In order to allow the running of Docker commands and the building of Docker images from inside the Docker 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
    }
  }
}
```

### 7.5.2. 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" ❷
    },
    {
      "secretSource": {
        "name": "secret2"
      },
      "mountPath": "/tmp/secret2"
    }
  ]
}
}
}

```

- ❶ **secretSource** is a reference to a secret in the same namespace as the build.
- ❷ **mountPath** is the path inside the custom builder where the secret should be mounted.

### 7.5.3. 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 ❶
    }
  }
}

```

- ❶ 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.

### 7.5.4. 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:

```

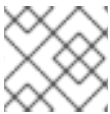
{
  "customStrategy": {
    ...
    "env": [

```

```
{
  "name": "HTTP_PROXY",
  "value": "http://myproxy.net:5187/"
}
```

## 7.6. 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
...
```

### 7.6.1. Using Private Repositories for Builds

Supply valid credentials to build an application from a private repository. Currently, only SSH key based authentication is supported. The repository keys are 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

For a SSH key to work in OpenShift builds, it must not have a passphrase set. 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 scmsecret ssh-privatekey=$HOME/.ssh/id_rsa
```



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 **scmsecret**:

```
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: "scmsecret"
        type: "Git"
  strategy:
    sourceStrategy:
      from:
        kind: "ImageStreamTag"
        name: "python-33-centos7:latest"
        type: "Source"
```

- 1 The URL of private repository is usually in the form **git@example.com: <username>/<repository>**.

## 7.7. 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
```

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

## 7.8. BINARY SOURCE

When the **BuildConfig.spec.source.type** is **Binary**, the build will expect a binary as input, and an inline Dockerfile is optional.

The binary is generally assumed to be a tar, gzipped tar, or zip file depending on the strategy. For **Docker** builds, this is the build context and an optional Dockerfile may be specified to override any Dockerfile in the build context. For **Source** builds, this is assumed to be an archive as described above. For **Source** and **Docker** builds, if **binary.asFile** is set the build will receive a directory with a single file. The **contextDir** field may be used when an archive is provided. Custom builds will receive this binary as input on standard input (**stdin**).

A binary source potentially extracts content, in which case **contextDir** allows changing to a subdirectory within the content before the build executes.

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

```
source:
  type: "Binary"
  binary: ❶
    asFile: "webapp.war" ❷
  contextDir: "app/dir" ❸
  dockerfile: "FROM centos:7\nRUN yum install -y httpd" ❹
```

- ❶ The **binary** field specifies the details of the binary source.
- ❷ The **asFile** field specifies the name of a file that will be created with the binary contents.
- ❸ The **contextDir** field specifies a subdirectory with the contents of a binary archive.
- ❹ If the optional **dockerfile** field is provided, it should be a string containing an inline Dockerfile that potentially replaces one within the contents of the binary archive.

## 7.9. 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 docker 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/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.

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

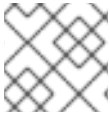
```
source:
  git:
    uri: https://github.com/openshift/ruby-hello-world.git
  images: ❶
    - from: ❷
      kind: ImageStreamTag
      name: myinputimage:latest
```

```

    namespace: mynamespace
  paths: ❸
  - destinationDir: injected/dir ❹
    sourcePath: /usr/lib/somefile.jar ❺
  - from:
    kind: ImageStreamTag
    name: myotherinputimage:latest
    namespace: myothernamespace
  pullSecret: mysecret ❻
  paths:
  - destinationDir: injected/dir
    sourcePath: /usr/lib/somefile.jar

```

- ❶ An array of one or more input images and files.
- ❷ A reference to the image containing the files to be copied.
- ❸ An array of source/destination paths.
- ❹ The directory relative to the build root where the build process can access the file.
- ❺ The location of the file to be copied out of the referenced image.
- ❻ 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.

## 7.10. USING SECRETS DURING A BUILD

In some scenarios operations performed by the build 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, you are building a NodeJS application and you 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, username and password. For security reasons, you do not want to expose your credentials in the application image.

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

### 7.10.1. Defining Secrets in the BuildConfig

First, you have to create the **Secret** you want to use. You can do that by executing the following command:

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

This command will create a new **Secret** named **secret-npmrc** and store the base64 encoded content of the **~/.npmrc** file in it.

Now, that you have the secret created, you can add references to **secrets** into the **source** section in the existing **BuildConfig**:

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

If you want to create a new **BuildConfig** and you want to include the secrets in it, you can 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 in both of these examples the **.npmrc** will be copied into a directory where the source code is located. In case of the OpenShift Source-To-Image builder images, this will be the image working directory which is set using the **WORKDIR** instruction in the **Dockerfile**. If you want to specify another directory, you can add a **destinationDir** into 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 **BuildConfig**:

```
$ 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 will be added into the **/etc** directory of the build environment. Note that for a [Docker strategy](#) the destination directory must be a relative path.

### 7.10.2. Source-to-Image Strategy

When you are using a **Source** strategy all defined source secrets will be copied to the respective **destinationDir**. If you left **destinationDir** empty then the secrets will be placed to the working directory of the builder image. The same rule is used when a destination directory is a relative path - the secrets will be placed into 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.

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

### 7.10.3. Docker Strategy

When you are using a **Docker** strategy, you can add all defined source secrets into your Docker image using the [ADD](#) and [COPY instructions](#) in your **Dockerfile**. If you don't 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 that for security reasons, users should always remove their secrets from the final application image. This removal should be part of the **Dockerfile** itself. In that case, the secrets will not be 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.

### 7.10.4. Custom Strategy

When you are using a **Custom** strategy, then all the defined source secrets will be available inside the builder container under **/var/run/secrets/openshift.io/build** directory. It is the custom build image's responsibility to use 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 **/var/run/secrets/openshift.io/build** directory or your builder can parse the **\$BUILD** environment variable which includes the full Build object serialized into JSON format.

## 7.11. STARTING A BUILD

Manually invoke a build using the following command:

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

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

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

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

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

## 7.12. CANCELING A BUILD

Manually cancel a build using the following command:

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

## 7.13. ACCESSING BUILD LOGS

To allow access to build logs, use the following command:

```
$ oc build-logs <buildName>
```

## 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 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 <i>assemble</i> 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.

## 7.14. SOURCE CODE

The source code location is one of the required parameters for the **BuildConfig**. The build uses this location and fetches the source code that is later built. The source code location definition is part of the **spec** section in the **BuildConfig**:

```
{
  "source" : {
```

```

    "type" : "Git", ❶
    "git" : { ❷
      "uri": "https://github.com/openshift/ruby-hello-world"
    },
    "contextDir": "app/dir", ❸
  },
}

```

- ❶ The **type** field describes which SCM is used to fetch your source code.
- ❷ 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 a branch name.
- ❸ 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.

## 7.15. 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](#)

### 7.15.1. Webhook Triggers

Webhook triggers allow you to trigger a new build by sending a request to the OpenShift 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** as part of the URL you supply to GitHub when configuring the webhook. The **secret** ensures that only you and your repository can trigger the build. The following example is a trigger definition JSON within the **BuildConfig**:

```

{
  "type": "GitHub",
  "github": {
    "secret": "secret101"
  }
}

```

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>/osapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/github

```

## 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** when defining the trigger, and the caller must provide this **secret** to trigger the build. The following is an example trigger definition JSON 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>/osapi/v1/namespaces/<namespace>/buildconfigs/<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.

### 7.15.2. 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 Docker 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.

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** part 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 Docker 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.

### 7.15.3. 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 JSON 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.

## 7.16. 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 Docker Registry, you don't have to do this because **secrets** are generated automatically for you by OpenShift.

The **.dockercfg** JSON file is found in your home directory by default and has the following format:

```
{
  "https://index.docker.io/v1/": { 1
    "auth": "YWRfbGZhcGU6R2labnRib21ifTE=", 2
    "email": "user@example.com" 3
  }
}
```

- 1 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**, 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:

```
$ 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": {
        "name": "private-image"
      },

```

```
        "pushSecret": {  
          "name": "dockerhub"  
        }  
      }  
    }  
  }  
}
```

4. Pull the builder Docker 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"  
  }  
}
```

## CHAPTER 8. DEPLOYMENTS

### 8.1. OVERVIEW

A deployment in OpenShift 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.

### 8.2. 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 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, ❷
"selector": {
    "name": "frontend"
},
"triggers": [
    {
        "type": "ConfigChange" ❸
    },
    {
        "type": "ImageChange", ❹
        "imageChangeParams": {
            "automatic": true,
            "containerNames": [
                "helloworld"
            ],
            "from": {
                "kind": "ImageStreamTag",
                "name": "origin-ruby-sample:latest"
            }
        }
    }
],
"strategy": { ❺
    "type": "Rolling"
}
}
}

```

- ❶ The replication controller template named **frontend** describes a simple Ruby application.
- ❷ There will be 5 replicas of **frontend** by default.
- ❸ A [configuration change trigger](#) causes a new deployment to be created any time the replication controller template changes.
- ❹ 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.
- ❺ The [Rolling strategy](#) is the default and may be omitted.

## 8.3. STARTING A DEPLOYMENT

To start a new deployment manually:

```
$ 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.

## 8.4. 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>
```

## 8.5. 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.

## 8.6. 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.

## 8.7. ROLLING BACK A DEPLOYMENT

Rollbacks revert an application back to a previous deployment and can be performed using the REST API or the CLI.

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
```

## 8.8. TRIGGERS

A deployment configuration can contain triggers, which drive the creation of new deployments in response to events, both inside and outside OpenShift.

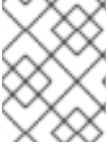
**WARNING**

If no triggers are defined on a deployment configuration, deployments must be [started manually](#).

### 8.8.1. Configuration Change Trigger

The ConfigChange trigger results in a new deployment whenever changes are detected to the replication controller template of the deployment configuration.





## NOTE

If a ConfigChange trigger is defined on a deployment configuration, the first deployment will be automatically created soon after the deployment configuration itself is created.

The following is an example of a ConfigChange trigger:

```
"triggers": [
  {
    "type": "ConfigChange"
  }
]
```

### 8.8.2. Image Change Trigger

The ImageChange trigger results in a new deployment whenever the value of an image stream tag changes.

The following is an example of an ImageChange trigger:

```
"triggers": [
  {
    "type": "ImageChange",
    "imageChangeParams": {
      "automatic": true, ①
      "from": {
        "kind": "ImageStreamTag",
        "name": "origin-ruby-sample:latest"
      },
      "containerNames": [
        "helloworld"
      ]
    }
  }
]
```

① If the **automatic** option 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 tag value differs from the current image specified in the deployment configuration's **helloworld** container, a new deployment is created using the new tag value for the **helloworld** container.

## 8.9. 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 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.

### 8.9.1. 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.

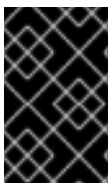
The following is an example of the Rolling strategy:

```
"strategy": {
  "type": "Rolling",
  "rollingParams": {
    "timeoutSeconds": 120, ❶
    "maxSurge": "20%", ❷
    "maxUnavailable": "10%" ❸
    "pre": {}, ❹
    "post": {}
  }
}
```

- ❶ How long to wait for a scaling event before giving up. Optional; the default is 120.
- ❷ **maxSurge** is optional and defaults to **25%**; see below.
- ❸ **maxUnavailable** is optional and defaults to **25%**; see below.
- ❹ **pre** and **post** are both [lifecycle hooks](#).

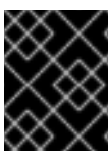
The Rolling strategy will:

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.

### 8.9.2. Recreate Strategy

The Recreate strategy has basic rollout behavior and supports [lifecycle hooks](#) for injecting code into the deployment process.

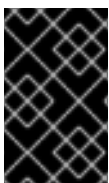
The following is an example of the Recreate strategy:

```
"strategy": {
  "type": "Recreate",
  "recreateParams": { 1
    "pre": {}, 2
    "post": {}
  }
}
```

- 1** **recreateParams** are optional.
- 2** **pre** and **post** are both [lifecycle hooks](#).

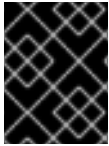
The Recreate strategy will:

1. Execute any "pre" lifecycle hook.
2. Scale down the previous deployment to zero.
3. Scale up the new deployment.
4. 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.



## IMPORTANT

When executing the "post" lifecycle hook, all failures will be ignored regardless of the failure policy specified on the hook.

### 8.9.3. 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** Docker 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 provides the following environment variables to the strategy process:

Environment Variable	Description
<b>OPENSIFT_DEPLOYMENT_NAME</b>	The name of the new deployment (a replication controller).
<b>OPENSIFT_DEPLOYMENT_NAMESPACE</b>	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.

## 8.10. 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:

<b>Abort</b>	The deployment should be considered a failure if the hook fails.
<b>Retry</b>	The hook execution should be retried until it succeeds.
<b>Ignore</b>	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.

#### 8.10.1. 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", ❶
        "command": [ ❷
          "/usr/bin/command", "arg1", "arg2"
        ],
        "env": [ ❸
          {
            "name": "CUSTOM_VAR1",
            "value": "custom_value1"
          }
        ]
      }
    }
  }
}
}
}
}
}
}
}

```

- ❶ The **helloworld** name refers to `spec.template.spec.containers[0].name`.
- ❷ This **command** overrides any **ENTRYPOINT** defined by the **openshift/origin-ruby-sample** image.
- ❸ **env** is an optional set of environment variables 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 container command will be **/usr/bin/command arg1 arg2**, and the hook container will have the **CUSTOM\_VAR1=custom\_value1** environment variable. Because the hook failure policy is **Abort**, the deployment will fail if the hook fails.

## 8.11. 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. Another way to limit resource use is to specify resource limits as part of the deployment strategy. In the following example, each of **resources**, **cpu**, and **memory** is optional.

```
{
```

```

    "type": "Recreate",
    "resources": {
      "limits": {
        "cpu": "100m", 1
        "memory": "256Mi" 2
      }
    },
  }
}

```

<sup>1</sup> **cpu** is in cpu units; **100m** represents 0.1 cpu units ( $100 * 1e-3$ )

<sup>2</sup> **memory** is in bytes; **256Mi** represents 268435456 bytes ( $256 * 2^20$ )

Deployment resources can be used with the **Recreate**, **Rolling**, or **Custom** deployment strategies.

## 8.12. MANUAL SCALING

In addition to rollbacks, you can exercise fine-grained control over the number of replicas 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**.

## CHAPTER 9. ROUTES

### 9.1. OVERVIEW

An OpenShift [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 router, or if using an unrelated host name you may need to modify its DNS records independently to resolve to the router.

### 9.2. CREATING ROUTES

Tooling for creating routes is developing. In the web console, they are displayed but there is not yet a method to create them. Using the CLI, you currently can create only an unsecured route:

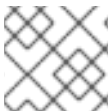
```
$ oc expose service/<name> --hostname=<www.example.com>
```

The new route inherits the name from the service unless you specify one.

#### Example 9.1. An Unsecured Route YAML Definition

```
apiVersion: v1
kind: Route
metadata:
  name: route-name
spec:
  host: www.example.com
  to:
    kind: Service
    name: service-name
```

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 must edit the unsecured route to add TLS termination.



#### NOTE

[TLS](#) is the replacement of SSL for HTTPS and other encrypted protocols.

```
$ oc edit route/<name>
```

#### Example 9.2. A Secure Route Using Edge Termination

```
apiVersion: v1
kind: Route
metadata:
  name: route-name
spec:
```



```

host: www.example.com
to:
  kind: Service
  name: service-name
tls:
  termination: edge
  key: |-
    -----BEGIN PRIVATE KEY-----
    [...]
    -----END PRIVATE KEY-----
  certificate: |-
    -----BEGIN CERTIFICATE-----
    [...]
    -----END CERTIFICATE-----
  caCertificate: |-
    -----BEGIN CERTIFICATE-----
    [...]
    -----END CERTIFICATE-----

```

You can specify a secured route without a key and certificate, in which case the [router's default certificate](#) will be used for TLS termination.



#### NOTE

TLS termination in OpenShift 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 10. INTEGRATING EXTERNAL SERVICES

### 10.1. OVERVIEW

Many OpenShift applications use external resources, such as external databases, or an external SaaS endpoint. These external resources can be modeled as native OpenShift services, so that applications can work with them as they would any other internal service.

### 10.2. 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 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 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 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 [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: {} ❶

```

- ❶ 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" ❶
subsets: ❷
-
  addresses:
  -
    IP: "10.0.0.0" ❸
  ports:
  -
    port: 3306 ❹
    name: "mysql"

```

- ❶ The name of the **Service** instance, as defined in the previous step.
- ❷ Traffic to the service will be load-balanced between the supplied **Endpoints** if more than one is supplied.
- ❸ 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).
- ❹ The **port** and **name** definition must match the **port** and **name** value in the service defined in the previous step.

3. 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: "${MYSQL_USER}" 2
    -
      name: "MYSQL_PASSWORD"
      value: "${MYSQL_PASSWORD}" 3
    -
      name: "MYSQL_DATABASE"
      value: "${MYSQL_DATABASE}" 4

```

- 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.

## 10.3. 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 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 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 [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>"
```

❶ Other fields on the **DeploymentConfig** are omitted.

❷ **SAAS\_API\_KEY**: The API key to use with the service.

❸ **SAAS\_USERNAME**: The user name to use with the service.

❹ **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 11. SECRETS

### 11.1. OVERVIEW

The **Secret** object type provides a mechanism to hold sensitive information such as passwords, OpenShift client config files, **dockercfg** files, 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.

```
{
  "apiVersion": "v1",
  "kind": "Secret",
  "metadata": {
    "name": "mysecret"
  },
  "namespace": "myns",
  "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](#).

### 11.2. 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.

#### 11.2.1. 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 [kubect1 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.

## 11.3. 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.

### 11.3.1. Creating Secrets

To create a secret object, use the following command, where the json file is a predefined secret:

```
$ oc create -f secret.json
```

### 11.3.2. Secrets in Volumes

See [Examples](#).

### 11.3.3. Image Pull Secrets

See the [Image Pull Secrets](#) topic for more information.

## 11.4. RESTRICTIONS

Secret volume sources are validated to ensure that the specified object reference points to a **Secret** object. Therefore, a secret needs to be created before the pods that depend on it.

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.

Currently, when mounting a secret, the service account for a pod must have the secret in the list of mountable secrets. If a template contains a secret definition and pods that consume it, the pods will be rejected until the service account is updated.

### 11.4.1. Secret Data Keys

Secret keys must be in a DNS subdomain.

## 11.5. EXAMPLES

### Example 11.1. YAML of a Pod Consuming Data in a Volume

```
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
```

## CHAPTER 12. IMAGE PULL SECRETS

### 12.1. OVERVIEW

[Docker registries](#) can be secured to prevent unauthorized parties from accessing certain images. If you are using OpenShift's integrated Docker registry and are pulling from image streams located in the same project, then your pod's service account should already have permissions and no additional action should be required. If this is not the case, then additional configuration steps are required.

### 12.2. INTEGRATED REGISTRY AUTHENTICATION AND AUTHORIZATION

OpenShift's integrated Docker registry authenticates using the same [tokens](#) as the OpenShift API. To perform a **docker login** against the integrated registry, you can choose any user name and email, but the password must be a valid OpenShift token. 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.

#### 12.2.1. Allowing Pods to Reference Images Across Projects

When using the integrated 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
```

### 12.3. ALLOWING PODS TO REFERENCE IMAGES FROM OTHER SECURED REGISTRIES

To pull a secured Docker image that is not from OpenShift's integrated registry, you must create a **dockercfg** secret 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>
```

If you do not already have a **.dockercfg** 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>
```

## CHAPTER 13. RESOURCE LIMITS

### 13.1. OVERVIEW

A **LimitRange** object enumerates the minimum and maximum resource usage values per object in a [project](#).

### 13.2. LIMITS

A **LimitRange** object can enumerate the following limits for each object type and resource value that is created or modified in the project. If a usage limit is defined, and the incoming resource exceeds the allowed range, then the resource is forbidden from the project.

For some fields, in the absence of a value on the incoming resource, it is possible to apply a default value, if it is specified in the **LimitRange** definition.

**Table 13.1. Limits**

Type	ResourceName	Description	Default Value Supported
Container	cpu	Minimum/maximum CPU allowed per container.	yes
Container	memory	Minimum/maximum memory allowed per container.	yes
Pod	cpu	Minimum/maximum CPU allowed across all containers in a pod.	no
Pod	memory	Minimum/maximum memory allowed across all containers in a pod.	no

### 13.3. LIMIT ENFORCEMENT

Once a **LimitRange** is created in a project, all resource create and modification requests are evaluated against each **LimitRange** object in the project. If the resource violates a minimum or maximum constraint enumerated, 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.

For example, if the container does not express a CPU resource requirement, but the **LimitRange** specifies a default value for container CPU, then the default value is set to the allowed CPU usage for that container, and the minimum (if specified) is set as the minimum requested value for that container.

#### Example 13.1. Limit Range Object Definition

```
{
  "apiVersion": "v1",
  "kind": "LimitRange",
  "metadata": {
    "name": "limits" ❶
  },
}
```

```

"spec": {
  "limits": [
    {
      "type": "Pod",
      "max": {
        "memory": "1Gi", 2
        "cpu": "2" 3
      },
      "min": {
        "memory": "1Mi", 4
        "cpu": "250m" 5
      }
    },
    {
      "type": "Container",
      "max": {
        "memory": "1Gi", 6
        "cpu": "2" 7
      },
      "min": {
        "memory": "1Mi", 8
        "cpu": "250m" 9
      },
      "default": {
        "memory": "1Mi", 10
        "cpu": "250m" 11
      }
    }
  ]
}

```

- 1 The name of the limit range document.
- 2 The maximum amount of memory that a pod can consume on a node across all containers.
- 3 The maximum amount of cpu that a pod can consume on a node across all containers.
- 4 The minimum amount of memory that a pod can consume on a node across all containers.
- 5 The minimum amount of cpu that a pod can consume on a node across all containers.
- 6 The maximum amount of memory that a single container in a pod can consume.
- 7 The maximum amount of cpu that a single container in a pod can consume.
- 8 The maximum amount of memory that a single container in a pod can consume.
- 9 The maximum amount of cpu that a single container in a pod can consume.
- 10 The default amount of memory that a container will request if not specified.
- 11 The default amount of cpu that a container will request if not specified.

## 13.4. CREATING A LIMIT RANGE

To apply a limit range to a project, create a [limit range object definition](#) on your file system to your specifications, then run:

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

## 13.5. VIEWING LIMITS

To view limits enforced in a project:

```
$ oc get limits
NAME
limits

$ oc describe limits limits
Name:          limits
Type           Resource      Min      Max      Default
----          -
Pod            memory        1Mi      1Gi      -
Pod            cpu           250m     2        -
Container      memory        1Mi      1Gi      1Mi
Container      cpu           250m     250m     250m
```

## 13.6. DELETING LIMITS

If you do not want to enforce limits in a project, you can remove any active limit range by name:

```
$ oc delete limits <limit_name>
```

## CHAPTER 14. RESOURCE QUOTA

### 14.1. OVERVIEW

A **ResourceQuota** object enumerates hard resource usage limits per project. It limits the total number of a particular type of object that may be created in a project, and the total amount of compute resources that may be consumed by resources in that project.

### 14.2. USAGE LIMITS

The following describes the set of limits that may be enforced by a **ResourceQuota**.

**Table 14.1. Usage limits**

Resource Name	Description
<b>cpu</b>	Total cpu usage across all containers
<b>memory</b>	Total memory usage across all containers
<b>pods</b>	Total number of pods
<b>replicationcontrollers</b>	Total number of replication controllers
<b>resourcequotas</b>	Total number of resource quotas
<b>services</b>	Total number of services
<b>secrets</b>	Total number of secrets
<b>persistentvolumeclaims</b>	Total number of persistent volume claims

### 14.3. QUOTA ENFORCEMENT

After a project quota 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.

Once a quota is created and usage statistics are up-to-date, 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. When you delete resources, a [configurable amount of time](#) determines how long it takes to reduce quota usage statistics to their current observed system value.

If project modifications exceed a quota usage limit, the server denies the action, and an appropriate error message is returned to the end-user explaining the quota constraint violated, and what their currently observed usage stats are in the system.



## 14.4. SAMPLE RESOURCE QUOTA FILE

resource-quota.json

```
{
  "apiVersion": "v1",
  "kind": "ResourceQuota",
  "metadata": {
    "name": "quota" ❶
  },
  "spec": {
    "hard": {
      "memory": "1Gi", ❷
      "cpu": "20", ❸
      "pods": "10", ❹
      "services": "5", ❺
      "replicationcontrollers": "5", ❻
      "resourcequotas": "1" ❼
    }
  }
}
```

- ❶ The name of this quota document
- ❷ The total amount of memory consumed across all containers may not exceed 1Gi.
- ❸ The total number of cpu usage consumed across all containers may not exceed 20 Kubernetes compute units.
- ❹ The total number of pods in the project
- ❺ The total number of services in the project
- ❻ The total number of replication controllers in the project
- ❼ The total number of resource quota documents in the project

## 14.5. CREATE A QUOTA

To apply a quota to a project:

```
$ oc create -f resource-quota.json
```

## 14.6. VIEW A QUOTA

To view usage statistics related to any hard limits defined in your quota:

```
$ oc get quota
NAME
quota
$ oc describe quota quota
Name:                quota
```

Resource	Used	Hard
-----	----	----
cpu	5	20
memory	500Mi	1Gi
Pods	5	10
replicationcontrollers	5	5
resourcequotas	1	1
services	3	5

## 14.7. CONFIGURING THE QUOTA SYNCHRONIZATION PERIOD

When a set of resources are deleted, the synchronization timeframe of resources is determined by the **resource-quota-sync-period** setting in the `/etc/openshift/master/master-config.yaml` file. Before your quota usage is restored, you may encounter problems when attempting to reuse the resources. Change the **resource-quota-sync-period** setting to have the set of resources regenerate at the desired amount of time (in seconds) and for the resources to be available again:

```
kubernetesMasterConfig:
  apiLevels:
    - v1beta3
    - v1
  apiServerArguments: null
  controllerArguments:
    resource-quota-sync-period:
      - "10s"
```

Adjusting the regeneration time can be helpful for creating resources and determining resource usage when automation is used.



### NOTE

The **resource-quota-sync-period** setting is designed to balance system performance. Reducing the sync period can result in a heavy load on the master.

## CHAPTER 15. MANAGING VOLUMES

### 15.1. 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.

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.

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.

### 15.2. 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 15.1. Object Selection**

Syntax	Description	Example
<b>&lt;object_type&gt; &lt;name&gt;</b>	Selects <b>&lt;name&gt;</b> of type <b>&lt;object_type&gt;</b> .	<b>deploymentConfig registry</b>
<b>&lt;object_type&gt;/&lt;name&gt;</b>	Selects <b>&lt;name&gt;</b> of type <b>&lt;object_type&gt;</b> .	<b>deploymentConfig/registry</b>
<b>&lt;object_type&gt; --selector=&lt;object_label_selector&gt;</b>	Selects resources of type <b>&lt;object_type&gt;</b> that matched the given label selector.	<b>deploymentConfig --selector="name=registry"</b>
<b>&lt;object_type&gt; --all</b>	Selects all resources of type <b>&lt;object_type&gt;</b> .	<b>deploymentConfig --all</b>
<b>-f or --filename=&lt;file_name&gt;</b>	File name, directory, or URL to file to use to edit the resource.	<b>-f registry-deployment-config.json</b>

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.

## 15.3. ADDING VOLUMES

To add a volume, a volume mount, or both to pod templates:

```
$ oc volume <object_type>/<name> --add [options]
```

**Table 15.2. Supported Options for Adding Volumes**

Option	Description	Default
<b>--name</b>	Name of the volume.	Automatically generated, if not specified.
<b>-t, --type</b>	Name of the volume source. Supported values: <b>emptyDir</b> , <b>hostPath</b> , <b>secret</b> , or <b>persistentVolumeClaim</b> .	<b>emptyDir</b>
<b>-c, --containers</b>	Select containers by name. It can also take wildcard '*' that matches any character.	'*'
<b>-m, --mount-path</b>	Mount path inside the selected containers.	
<b>--path</b>	Host path. Mandatory parameter for <b>--type=hostPath</b> .	
<b>--secret-name</b>	Name of the secret. Mandatory parameter for <b>--type=secret</b> .	
<b>--claim-name</b>	Name of the persistent volume claim. Mandatory parameter for <b>--type=persistentVolumeClaim</b> .	
<b>--source</b>	Details of volume source as a JSON string. Recommended if the desired volume source is not supported by <b>--type</b> . See <a href="#">available volume sources</a>	
<b>-o, --output</b>	Display the modified objects instead of updating them on the server. Supported values: <b>json</b> , <b>yaml</b> .	
<b>--output-version</b>	Output the modified objects with the given version.	<b>api-version</b>

## 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"
  } }'
```

## 15.4. 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
```

## 15.5. REMOVING VOLUMES

To remove a volume or volume mount from pod templates:

```
$ oc volume <object_type>/<name> --remove [options]
```

**Table 15.3. Supported Options for Removing Volumes**

Option	Description	Default
<b>--name</b>	Name of the volume.	
<b>-c, --containers</b>	Select containers by name. It can also take wildcard '*' that matches any character.	'*'
<b>--confirm</b>	Indicate that you want to remove multiple volumes at once.	
<b>-o, --output</b>	Display the modified objects instead of updating them on the server. Supported values: <b>json</b> , <b>yaml</b> .	
<b>--output-version</b>	Output the modified objects with the given version.	<b>api-version</b>

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
```

## 15.6. 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
<b>--name</b>	Name of the volume.	

Option	Description	Default
<b>-c, --containers</b>	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
```

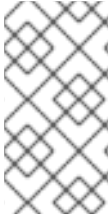
List volume **v1** defined on all deployment configurations:

```
$ oc volume dc --all --name=v1
```

## CHAPTER 16. USING PERSISTENT VOLUMES

### 16.1. OVERVIEW

A **PersistentVolume** object is a storage resource in an OpenShift cluster. Storage is provisioned by an administrator by creating **PersistentVolume** objects from sources such as GCE Persistent Disks, AWS Elastic Block Stores (EBS), and NFS mounts.



#### NOTE

Persistent volume plug-ins other than the supported [NFS](#) plug-in, such as [AWS Elastic Block Stores \(EBS\)](#), [GCE Persistent Disks](#), [GlusterFS](#), [iSCSI](#), and [RADOS \(Ceph\)](#), are currently in [Technology Preview](#). The Administrator Guide provides instructions on provisioning an OpenShift cluster with [persistent storage using NFS](#).

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.

### 16.2. REQUESTING STORAGE

You can request storage by creating **PersistentVolumeClaim** objects in your projects:

#### Example 16.1. Persistent Volume Claim Object Definition

```
{
  "apiVersion": "v1",
  "kind": "PersistentVolumeClaim",
  "metadata": {
    "name": "claim1"
  },
  "spec": {
    "accessModes": [ "ReadWriteOnce" ],
    "resources": {
      "requests": {
        "storage": "5Gi"
      },
      "volumeName": "pv0001"
    }
  }
}
```

### 16.3. 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 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    RWO
Bound      yournamespace / claim1
```

## 16.4. CLAIMS AS VOLUMES IN PODS

A **PersistentVolumeClaim** is used by a pod as a volume. OpenShift 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.

### Example 16.2. 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"
      }
    }]
  }
}
```

See [Kubernetes Persistent Volumes](#) for more information.

## CHAPTER 17. EXECUTING REMOTE COMMANDS

### 17.1. 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.

### 17.2. 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
```

### 17.3. 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 18. PORT FORWARDING

### 18.1. 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.

### 18.2. BASIC USAGE

Support for port forwarding is built into [the CLI](#):

```
$ oc port-forward -p <pod> [<local_port>:]<pod_port> [[<local_port>:]<pod_port> ...]
```

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:

<b>5000</b>	The client listens on port 5000 locally and forwards to 5000 in the pod.
<b>6000:5000</b>	The client listens on port 6000 locally and forwards to 5000 in the pod.
<b>:5000</b> or <b>0:5000</b>	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 -p mypod 5000 6000
```

To listen on port **8888** locally and forward to **5000** in the pod, run:

```
$ oc port-forward -p mypod 8888:5000
```

To listen on a free port locally and forward to **5000** in the pod, run:

```
$ oc port-forward -p mypod :5000
```

Or, alternatively:

```
$ oc port-forward -p mypod 0:5000
```

### 18.3. 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 19. SHARED MEMORY

### 19.1. 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.

### 19.2. 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
```

- ❶ specifies the tmpfs volume **dshm**.
- ❷ enables POSIX shared memory for **hello-container1** via **dshm**.
- ❸ 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 20. APPLICATION HEALTH

### 20.1. 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 applications have a number of options to detect and handle unhealthy containers.

### 20.2. 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 <b><code>template.spec.containers.livenessprobe</code></b> 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 <b><code>template.spec.containers.readinessprobe</code></b> stanza of a pod configuration.

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 20.1. 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 20.2. 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 20.3. Liveness TCP Socket Check**

```
...
livenessProbe:
  tcpSocket:
    port: 8080
    initialDelaySeconds: 15
    timeoutSeconds: 1
...
```

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 21. EVENTS

### 21.1. OVERVIEW

OpenShift events are modeled based on events that happen to API objects in an OpenShift cluster. Events allow OpenShift 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.

### 21.2. QUERYING EVENTS WITH THE CLI

This section will be expanded as more information becomes available

### 21.3. VIEWING EVENTS IN THE CONSOLE

This section will be expanded as more information becomes available

### 21.4. OUT-OF-BAND INTERFACE

OpenShift administrators can consume events directly via an out-of-band interface instead of querying the API server. This can be helpful, for example, if the API server cannot be contacted by a system component in order to record events.

This section will be expanded as more information becomes available

## CHAPTER 22. DOWNWARD API

### 22.1. 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. Containers can consume information from the downward API [using environment variables](#) or a [volume plug-in](#).

### 22.2. SELECTING FIELDS

Fields within the pod are selected using the **FieldRef** API type. **FieldRef** has two fields:

Field	Description
<b>fieldPath</b>	The path of the field to select, relative to the pod.
<b>apiVersion</b>	The API version to interpret the <b>fieldPath</b> selector within.

Currently, there are four valid selectors in the v1 API:

Selector	Description
<b>metadata.name</b>	The pod's name.
<b>metadata.namespace</b>	The pod's namespace.
<b>metadata.labels</b>	The pod's labels.
<b>metadata.annotations</b>	The pod's annotations.

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.

### 22.3. 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.

## 22.4. 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 22.1. Downward API Volume Plug-in Configuration

```
spec:
  volumes:
    - name: podinfo
      metadata: ❶
        items: ❷
          - name: "labels" ❸
            fieldRef:
              fieldPath: metadata.labels ❹
```

- ❶ The **metadata** field of the volume source configures the downward API volume.
- ❷ The **items** field holds a list of fields to project into the volume.
- ❸ The name of the file to project the field into.
- ❹ 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 23. MANAGING ENVIRONMENT VARIABLES

### 23.1. OVERVIEW

You can set, unset or list environment variables in pods or pod [templates](#) using the **oc env** command.

### 23.2. CLI INTERFACE

OpenShift provides the **oc 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.

The **oc env** command uses the following general syntax:

```
$ oc env <object-selection> <environment-variables> [options]
```

There are several ways to express **<object-selection>**.

Syntax	Description	Example
<b>&lt;object-type&gt; &lt;object-name&gt;</b>	Selects <object-name> of type <object-type>.	<b>dc registry</b>
<b>&lt;object-type&gt;/&lt;object-name&gt;</b>	Selects <object-name> of type <object-type>.	<b>dc/registry</b>
<b>&lt;object-type&gt; --selector=&lt;object-label-selector&gt;</b>	Selects objects of type <object-type> that match <object-label-selector>.	<b>dc --selector="name=registry"</b>
<b>&lt;object-type&gt; --all</b>	Selects all objects of type <object-type>.	<b>dc --all</b>
<b>-f, --filename=&lt;ref&gt;</b>	Look in <ref>—a filename, directory name, or URL—for the definition of the object to edit.	<b>-f registry-dc.json</b>

Supported common options for set, unset or list environment variables:

Option	Description
<b>-c, --containers [&lt;name&gt;]</b>	Select containers by <name>. You can use asterisk (*, U+2A) as a wildcard. If unspecified, <name> defaults to *.
<b>-o, --output &lt;format&gt;</b>	Display the changed objects in <format>—either <b>json</b> or <b>yaml</b> —instead of updating them on the server. This option is incompatible with <b>--list</b> .

Option	Description
<b>--output-version &lt;api-version&gt;</b>	Output the changed objects with <b>&lt;api-version&gt;</b> instead of the default API version.
<b>--resource-version &lt;version&gt;</b>	Proceed only if <b>&lt;version&gt;</b> matches the current resource-version in the object. This option is valid only when specifying a single object.

## 23.3. SET ENVIRONMENT VARIABLES

To set environment variables in the pod templates:

```
$ oc env <object-selection> KEY_1=VAL_1 ... KEY_N=VAL_N [<set-env-options>]
[<common-options>]
```

Set environment options:

Option	Description
<b>-e, --env=&lt;KEY&gt;=&lt;VAL&gt;</b>	Set given key value pairs of environment variables.
<b>--overwrite</b>	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 env dc/registry STORAGE=/data
$ oc 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 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**.

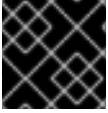
```
$ oc env -f rc.json STORAGE=/opt -o yaml > rc.yaml
```

## 23.4. UNSET ENVIRONMENT VARIABLES

To unset environment variables in the pod templates:

```
$ oc 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 env dc/d1 ENV1- ENV2-
```

This removes environment variable **ENV** from all replication controllers:

```
$ oc env rc --all ENV-
```

This removes environment variable **ENV** from container **c1** for replication controller **r1**:

```
$ oc env rc r1 --containers='c1' ENV-
```

## 23.5. LIST ENVIRONMENT VARIABLES

To list environment variables in pods or pod templates:

```
$ oc env <object-selection> --list [<common-options>]
```

This example lists all environment variables for pod **p1**:

```
$ oc env pod/p1 --list
```

## CHAPTER 24. REVISION HISTORY: DEVELOPER GUIDE

### 24.1. WED MAY 25 2016

Affected Topic	Description of Change
<a href="#">Templates</a>	Fixed <b>oc process</b> example in the <a href="#">Generating a List of Objects</a> section.

### 24.2. THU MAY 19 2016

Affected Topic	Description of Change
<a href="#">Builds</a>	Updated the examples in the <a href="#">Defining a BuildConfig</a> , <a href="#">Source Code</a> , and <a href="#">Using a Proxy for Git Cloning</a> sections to use https for GitHub access.

### 24.3. TUE APR 19 2016

Affected Topic	Description of Change
<a href="#">Developer Guide → Authentication</a>	Added a pointer to the <a href="#">browser requirements</a> from the <a href="#">Web Console Authentication</a> section.

### 24.4. THU MAR 17 2016

Affected Topic	Description of Change
<a href="#">Builds</a>	In the <a href="#">No Cache</a> subsection of the <a href="#">Docker Strategy Options</a> section, spelled <b>noCache</b> correctly in the example.

### 24.5. MON FEB 15 2016

Affected Topic	Description of Change
<a href="#">Integrating External Services</a>	Corrected port numbers across the page and converted examples to YAML.

### 24.6. TUE JUN 23 2015

OpenShift Enterprise 3.0 release.

